Introduction

Goal

To make an entertainment website that incorporates an sql database and a working backend and front end to deliver a final result.

Purpose

The purpose of this digital design outcome is to create a functional application that delivers an interactive website that consists of a flask backend with an sql database. The design will implement advanced programming / development techniques like CSS and backend to frontend communication. The main point of the website's existence is to entertain the user through the use of a game.

Outline

My plan for the digital outcome will consist of a website that's purpose is to entertain the user as a game website. The website will consist of a game for the user to play, and a leaderboard where other users can make an account and look at other users on the leaderboard. The website will have a filtering system on the statistics page where users can have control of what they want to search for specifically.

Github source code link

https://github.com/alexanderthegreatalmightypowerful/FlaskApp

Planning

After encountering so many 404 pages on github and other websites when looking for something, I just can't help but get frustrated. So in retaliation, I came up with a website that uses html elements to create objects and used javascript to give those objects life. I then am planning to use Sql and python to make a flask server that can retrieve the users statistics (like hits and wins) and store them. The home page has a scoreboard that can be sorted by queries and searches. Besides being a fun website to play on, it really is just a website built out of anger. So in short, the whole purpose is to entertain bored gamers by using a game as the entertainer.

The main part of the website, the bullet-hell game, will consist of a boss battle where you click on the boss to take down its health points. Once the boss is down to 0 hit points, the boss will move onto the next stage, and start to deal attacks that get harder and harder to

dodge. The website's theme is a website errors, so there will be references to website error codes and glitches that can be found when browsing the web (like google not finding your results or github error cat showing up to attack the players mouse). The player's hit points are directly linked to their mouse so all attacks will be directed towards it. This game-like strategy was inspired by JSaB and Osu which are both bullet hell games.

Requirements

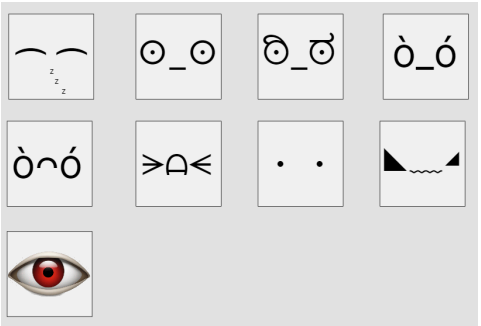
- World statistics page
- Private profile manipulation / page
- Game page
- Account creation Page
- Account sign in Page
- About Page
- Front end requests with js
- Back End Flask
- Easily navigable

Tools

- Flask - python routing library
- Visual Studio Code - for coding
- Flask-Session - user session library for flask
- GitHub - code version storage
- SQL Lite Studio - editing / viewing database
- Lucid Charts - making ER diagrams

Mock Website Designs

This mock design was made using Lunacy which is a website design application



Game Boss Face designs (Also Made in Lunacy)

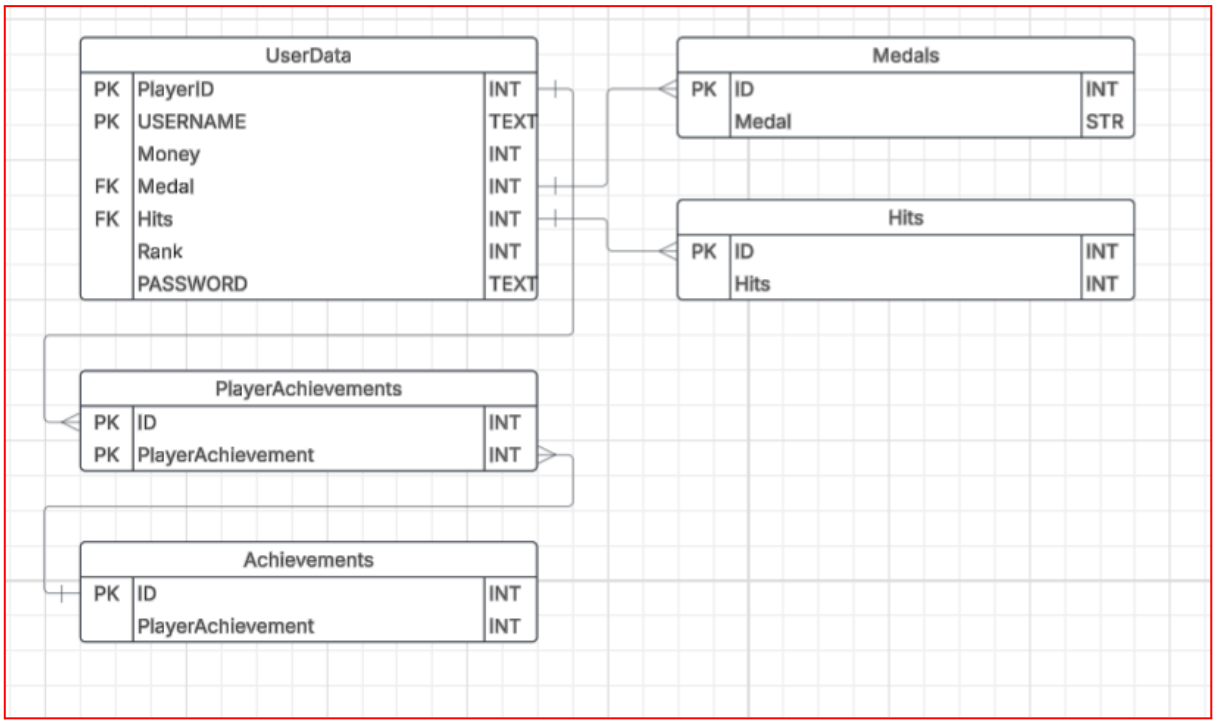
Database Plan

In my website, I will have a main table that holds the username, password and stats of the player. Attached to the table, there will be two foreign key tables, one for medals (which is awarded to how many times the player beats the game) and a ‘hits’ table (that is stored how many hits the player did). The other two tables are ‘many to many’ tables. This is where achievements are stored. Every player can have multiple achievements and achievements can be linked to many players. On the homepage of the website, there will be direct access to the leader board where a user can sort the table to display, for example, what players have completed the game 10 times.

SQLite3 Database

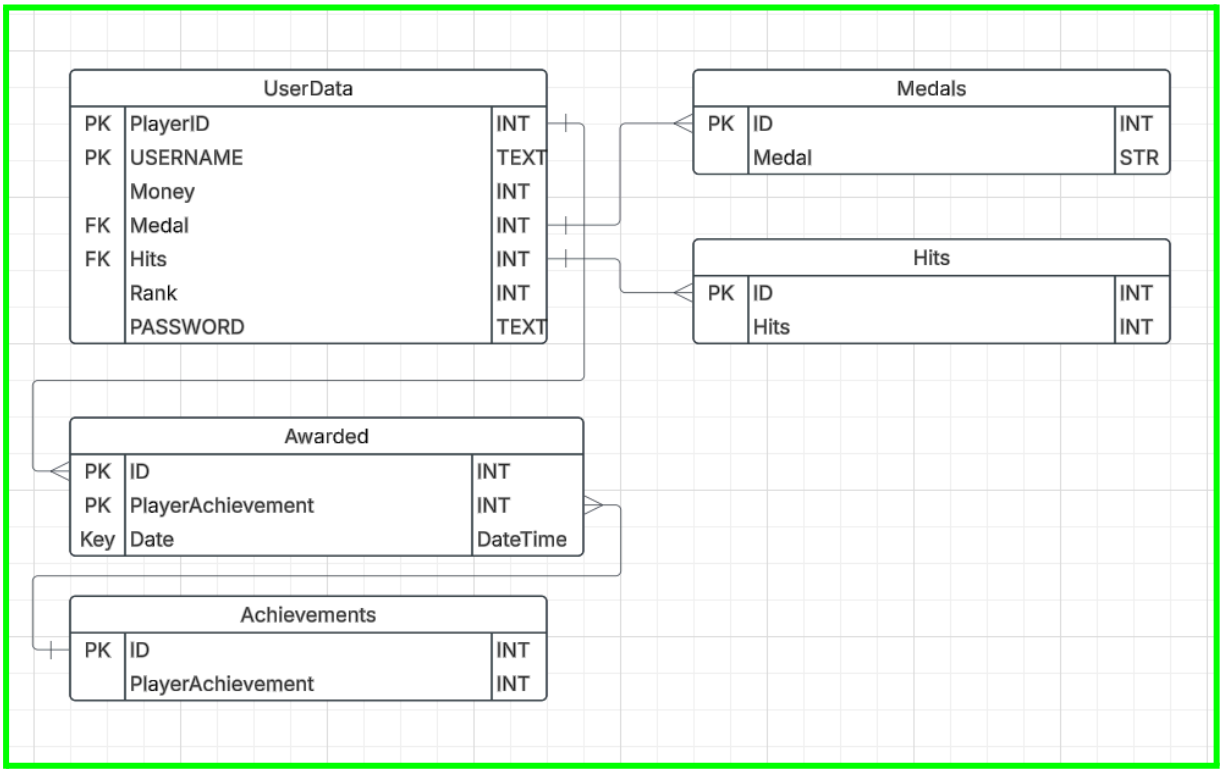
Database Iterations

First Iteration



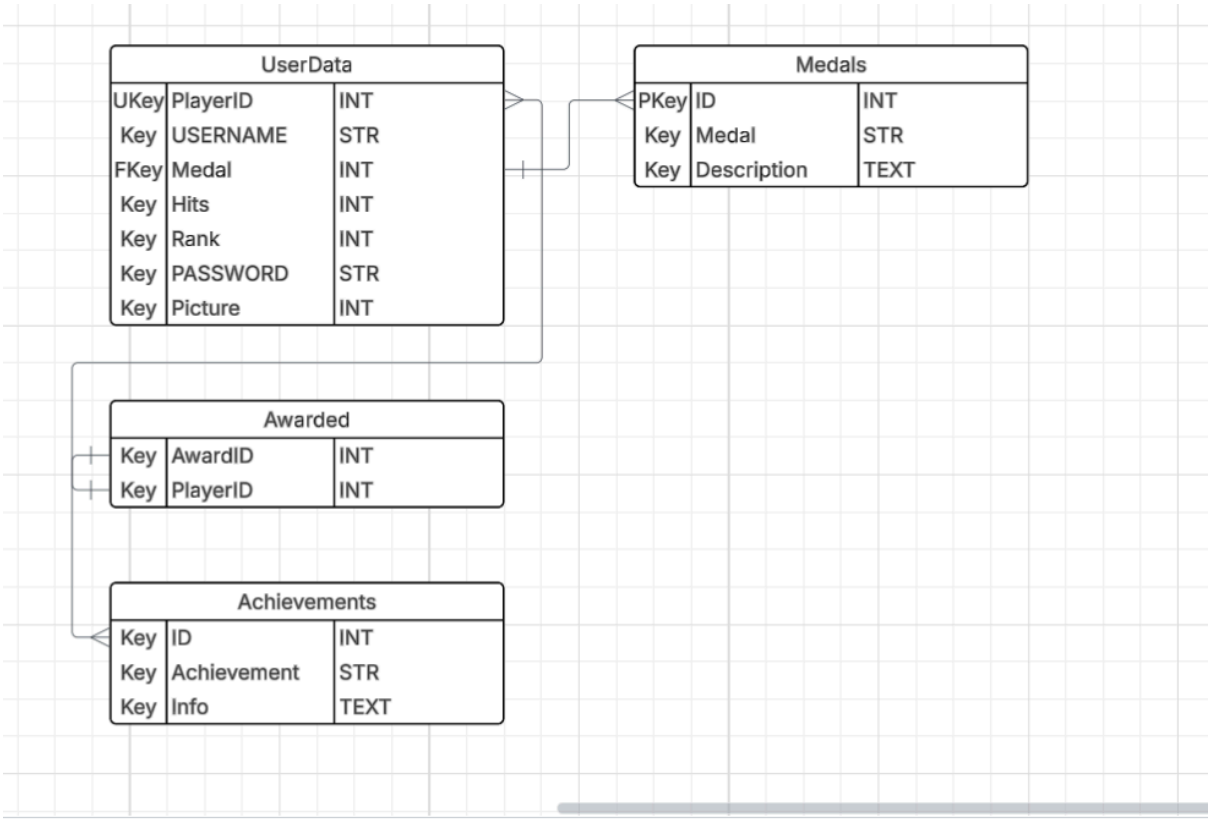
In the first iteration, I only had a basic idea of what I wanted my game and website to be about. I wanted a many to many table to make an achievements system, and a join table which is the medals and hits amount where i could sort people by how many hits and medals they have. The main point of the database is to store user data and information of what they have done on the website like how many times they defeated the boss or how many times they clicked on the boss.

Second Iteration



The second iteration wasn't much of a change but I added a date to when a player gained an achievement. This could create a better option for a future filtering system where we can sort by who first got the achievement and have a ranking system based on that.

Final Iteration

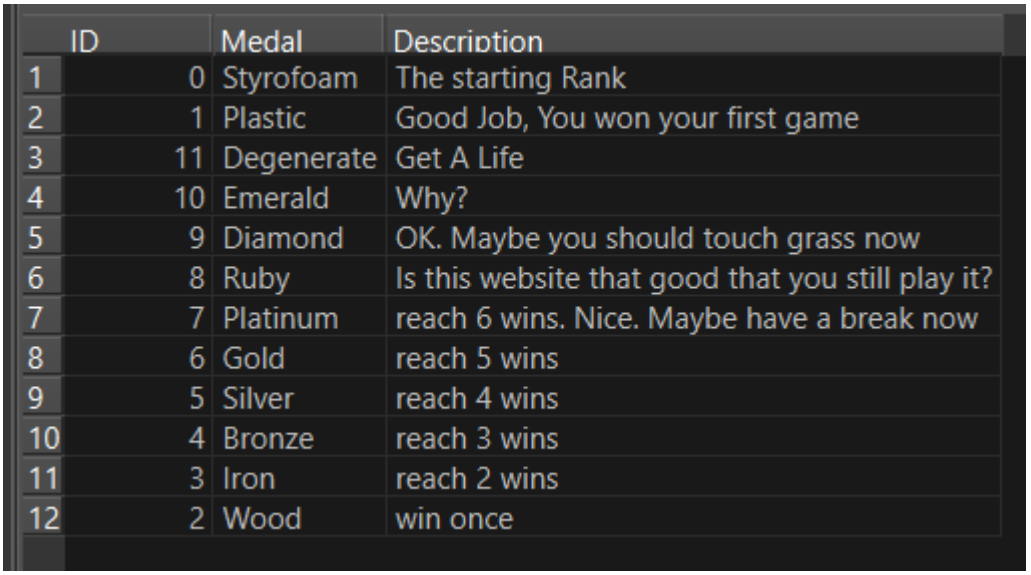


Once I knew exactly what my website would be about, I concluded that I no longer needed some variable as seen in the previous iterations which were: Money and the Date on the awarded table. I also removed the Hits Foreign key table as the sorting could be done in python and not by sql.

Initial Data

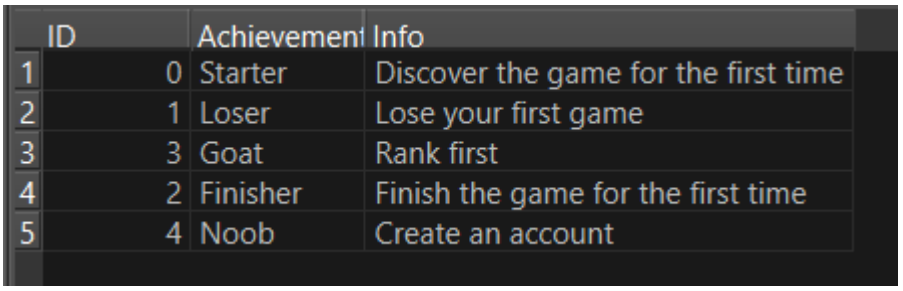
The only tables that will have any initial data will be the medals and achievements tables as the other tables will be filed later once accounts have been made.

Medals Table



	ID	Medal	Description
1	0	Styrofoam	The starting Rank
2	1	Plastic	Good Job, You won your first game
3	11	Degenerate	Get A Life
4	10	Emerald	Why?
5	9	Diamond	OK. Maybe you should touch grass now
6	8	Ruby	Is this website that good that you still play it?
7	7	Platinum	reach 6 wins. Nice. Maybe have a break now
8	6	Gold	reach 5 wins
9	5	Silver	reach 4 wins
10	4	Bronze	reach 3 wins
11	3	Iron	reach 2 wins
12	2	Wood	win once

Achievements Table



	ID	Achievement	Info
1	0	Starter	Discover the game for the first time
2	1	Loser	Lose your first game
3	3	Goat	Rank first
4	2	Finisher	Finish the game for the first time
5	4	Noob	Create an account

SQLite3 Python Backend

```
def read_sql_file_for_tables(file) -> list:···

class SqlDatabase():
    """
    Since the website is primarly built for a database, to make it easier to load data through many functions
    we can use a self cleaning up class that simplifies and takes care of systems like cleanup.
    """
    dumps = [] #public class object for active database classes (used for cleanup)
    def __init__(self, path = 'GameData.db'): #what will first happened when the class object is created
        total_path = os.getcwd() + '\\DataBases\\' + path
        print('THE TOTAL PATH:',total_path)
        self.tables = []
        try:
            self.connection = sql.connect(total_path)
            self.db = self.connection.cursor()
            self.closed = False
            self.tables = read_sql_file_for_tables(total_path)
        except Exception as e:
            self.closed = True
            print('SQL FAILED TO CONNECT:', e)

        for item in SqlDatabase.dumps:
            #item.close()
            del item #clean up existing active unused database classes
        SqlDatabase.dumps = []
        SqlDatabase.dumps.append(self)

    def fetchall(self, command = None, close = True) -> tuple: #fetches every value returned from the sql querey···

    def fetchone(self, command = None, close = True) -> tuple: #fetches the first value from querey···

    def execute(self, command = None, close = True): #execute command allows any querey to be executed.···

    def close(self): #closes database (can be called manualy or automaticly by querey functions)···
```

To simplify the process of pushing retrieving data from the database, I made a class that holds various functions to speed up production.

```
    dumps = [] #public class object for active database classes (used for cleanup)
    def __init__(self, path = 'GameData.db'): #what will first happened when the class object is created
        total_path = os.getcwd() + '\\DataBases\\' + path
        print('THE TOTAL PATH:',total_path)
        self.tables = []
        try:
            self.connection = sql.connect(total_path)
            self.db = self.connection.cursor()
            self.closed = False
            self.tables = read_sql_file_for_tables(total_path)
        except Exception as e:
            self.closed = True
            print('SQL FAILED TO CONNECT:', e)

        for item in SqlDatabase.dumps:
            if item.closed == False:
                item.close()
            del item #clean up existing active unused database classes
        SqlDatabase.dumps = []
        SqlDatabase.dumps.append(self)
```

When first creating the class object, it will go through an automation phase. It will first connect to the database chosen (default is the user database), open a connection to it, and close any other detected opened databases which are stored in a public class list. This makes keeping track of opened and closed databases a breeze and less of a worry.

```python
def fetchall(self, command = None, close = True) -> tuple: #fetches every value returned from the sql querey
    """
    in most cases, we only need to use one command to get what we want from the database
    so the close varaibale is set to 'True' by default so when you call function sto extract or append data
    to the database, it will close the tunnel automaticly (if close's value is unchanged).
    """

    if self.closed == True:
        return []

    if command == None:
        return
    if self.close == True:
        self.close()
    return self.db.execute(command).fetchall()
```

```python
def fetchone(self, command = None, close = True) -> tuple: #fetches the first value from querey···

def execute(self, command = None, close = True): #execute command allows any querey to be executed.···

def close(self): #closes database (can be called manualy or automaticly by querey functions)···
```

The first function fetches all results from a query. The point of these functions is to keep the code clean and more simple. All functions that connect to the database through a query get automatically closed by default unless the function is specified to not close the connection through the close key variable. The rest of the functions do different tasks with a connection to the database.

```python
def update_sql(table, name, value, where, what) -> None: #update something in the database
    if type(value) == str: #we add extra parenthasies so when it is formated in string, it keeps its
        value = f"'{value}'"
    if type(what) == str:
        what = f'"{what}"'
    base = SqlDatabase()
    base.execute(f'UPDATE {table} SET {name} = {value} WHERE {where} = {what}')
```

For updating a single column in the database, we use this function that can automatically parse and push data through arguments. The ‘table’ asks for the table name, the ‘name’ is the column name, the ‘value’ is what you want to change, and the ‘where’ and ‘what’ are conditions.

```python
def check_sql_data(data:str) -> bool: #check for malisouse inputs
    splitz = data.lower().split(' ')
    if "insert" in splitz:
        return False
    if '*' in list(data):
        return False
    if 'password' in splitz:
        return False

    return True
```

For malicious input, we can make a simple filtering system where we can search for key words that should not be allowed through the execution function with the database pointer.

```python
def organize_sql_data(data) -> dict:
    """
    like the function suggests, we organize a returned sql querey from the SqlDatabase class.
    To easily make display tables in html, we have to send a readible, easily interperated dictionary,
    where the javascript in the front end can convert the organised data in clean html tables.
    """

    try:
        result = data
        columns = [i[0] for i in SqlDatabase.dumps[-1].db.description] #exctract columns from sql class

        #===============TABLES
        tables = {}

        #END TABLES

        dump = SqlDatabase.dumps[-1]
        table_names = SqlDatabase.dumps[-1].tables
        print("DATABASE TABLES:", table_names)

        for item in table_names:
            tables[item] = []
            for keys in dump.db.execute(f"SELECT * FROM pragma_table_info('{item}');").fetchall():
                tables[item].append(keys[1])

        #print(tables)

        print_line = ''

        data = {}
        table_rows = []
        column_data = []

        for column in columns:
            print_line += f'{str(column)}{" " * (15 - len(str(column)))} | '
            data[str(column)] = []
            column_data.append(str(column))

        print_line += f'\n{("_" * 18) * len(columns)}\n' #prepares a string that will be printed into the co
        colum_name = -1
        for item in result:
            table_rows.append([])
            colum_name += 1
            for words in item:
                print_line += f'{str(words)}{" " * (15 - len(str(words)))} | '
                table_rows[colum_name].append(str(words))

            print_line += '\n'


        return {'columns' : columns, 'data' : data, 'rows' : table_rows, 'output' : print_line, 'tables' : t
    except Exception as e: # if an error, still send something else it will break. we send N/A here
        print(e)
        return {'columns' : ['N/A'], 'data' : 'there is no data', 'rows' : {'N/A':'N/A'},'output' : '', 'tab
```

To allow the frontend to understand the structure of the fetched database data, I made a simple sorter that creates a dictionary that simplifies and makes the data more structured and prepared for inputting into an html table.

Database Relevant Implications

Privacy

When developing a database for a website, it is important to hide the raw data and keep user information safe from hackers and public view. This means that I must keep making the program only allow the public to view what I want them to view. This means only showing people public information like the usernames and statistics, while keeping things like passwords hidden.

When a user creates an account on a website, it is important to respect their privacy. To make sure that I, myself the creator of the website, don't obstruct a user's privacy, I decided to hash the passwords. By hashing passwords, if there would be a database leak, the passwords would be safely hidden from public eyes. The website then compares hashes when signing into the website which is safer than comparing the raw passwords.

By also not asking users for private information like birth dates and email addresses, there is no sensitive information stored onto the database. This means that the database can technically be publicly seen and have no effect on users outside of the website.

Social

Social as a relevant implication refers to how end users see and interact with the database. This means storing data that changes as a user interacts with a website and sharing that data live with other users that have access to see the data.

When developing the database, it was important to consider what is important to share, store and update. For the game, the website does store when the user clicks and what stage they are in the game. However, it does this only in the game.The data collected purely for keeping track of their users progress with the website's game and the data is not exploited to do anything like marketing or data selling. All the data that is collected during the game is meant to be viewed by the public and is not intended to be harmful and is not harmful.

The data is also shared to other users through the filtering menu. The user can input buttons which will generate a querey that will be sent to the database and then return to the front end for the user to see. So the user has control and freedome to query the database as they want through the filtering options.

Legal

Legal as a relevant implication means to make sure that the database abides by consumer rights and the law. This means that the database is not storing anything ilegal or not by the users permission.

Like brefiely mentioned in the social tab, the database doest not store emails and sell the users data. The database also makes sure to have public seeing limits meaning the users are limited to what they can see on a specific public account. In my case, they dont see the hashed password or the user ID. The table names and queries are also hidden from the public so they cant do malicious injections into the database.

Database Integrity & Testing

To ensure that the front end properly passes the sign in information to the backend then into the database, I have concluded some tests to show that the database is storing the correct data received from the frontends input.

Account creation

In this test, I will create an account and compare the database before and after the creation.

	PlayerID	USERNAME	Money	Medal	Hits	Rank	PASSWORD	Wins	Picture
1	1	alex	20	11	2884	1	81dc9bdb52d04dc20036dbd8313ed055	0	8
2	2	INSERT	20	0	328	2	e10adc3949ba59abbe56e057f20f883e	0	8
3	3	22197	20	0	0	4	cf9bb12aa47ab3e8ce353b001b8b79e4	0	0
4	4	22343	20	0	0	5	91f724a7a4de4b13f12e65ce67ed0e23	0	0
5	5	22177	20	0	0	6	e10adc3949ba59abbe56e057f20f883e	0	0
6	6	22146	20	0	0	7	358aaf4a91006722f704e432b82fa60e	0	0
7	7	hi_there	20	0	152	3	e10adc3949ba59abbe56e057f20f883e	0	4
8	8	Aaronatedih	20	0	0	8	cb9fbb82698d4d941edb1db8c6796fe7	0	1

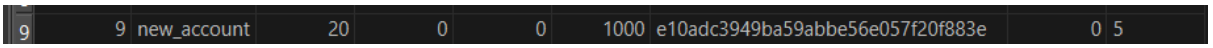
Before

We expect that when the account is created, we will see a new name and player id appearing in the database. In addition, we should also see the player get awarded the ‘Noob’ achievement so we should see the playerID 9 added to the many to many table with the AwardID 4 (the noob achievement id).

	AwardID	PlayerID
1	0	1
2	4	2
3	4	3
4	4	4
5	1	1

After

	PlaverID	USERNAME	Monev	Medal	Hits	Rank	PASSWORD	Wins	Picture
1	1	alex	20	11	2884	1	81dc9bdb52d04dc20036dbd8313ed055	0	8
2	2	INSERT	20	0	328	2	e10adc3949ba59abbe56e057f20f883e	0	8
3	3	22197	20	0	0	4	cf9bb12aa47ab3e8ce353b001b8b79e4	0	0
4	4	22343	20	0	0	5	91f724a7a4de4b13f12e65ce67ed0e23	0	0
5	5	22177	20	0	0	6	e10adc3949ba59abbe56e057f20f883e	0	0
6	6	22146	20	0	0	7	358aaf4a91006722f704e432b82fa60e	0	0
7	7	hi_there	20	0	152	3	e10adc3949ba59abbe56e057f20f883e	0	4
8	8	Aaronatedih	20	0	0	8	cb9fbb82698d4d941edb1db8c6796fe7	0	1
9	9	new_account	20	0	0	1000	e10adc3949ba59abbe56e057f20f883e	0	5



New Account:

	AwardID	PlaverID
1	0	1
2	4	2
3	4	3
4	4	4
5	1	1
6	4	5
7	4	6
8	4	7
9	1	7
10	4	8
11	4	9

We can see that the new account got added to the table correctly.

Flask & Flask Session Backend

```python
@app.errorhandler(404)  # handle 404 missing page errors
def page_not_found(e):
    return render_template('404.html', title='Game'), 404


@app.route('/')
@app.route('/home')  # home route / front page
def home():
    global session
    print(session)
    print(session, app.session_interface.get_cookie_path(app))
    print(app.session_interface.get_cookie_name(app=app))

    try:
        print("COOKIE USER!", get_user_by_cookie(request, session))
    except Exception as e:
        print(e)

    return render_template('home.html', title='Home')
```

When creating the base backend, we need to start with a homepage and an error page. When the user first enters the website, the routes “/home” and “/” route the user to the home page. The “/” just means that if the user's url doesn't include any subsites, they will be redirected to the home site.

```python
def jsonify_request(data) -> list: #this handles incoming ajax requests and parses them to a
    js = json.loads(data.decode())
    return js['data']
```

When receiving data from the frontends javascript, We have to convert the stringified json data into a usable dictionary for python to understand. This function instals the json string and returns a dictionary.

```python
ef get_cookie(request) -> str:  # returns cookie object from front end
   return request.cookies.get('session')


ef get_user_by_cookie(request, session) -> str:  # get cookie by username
   cookie = get_cookie(request)
   return session[cookie]
```

For the client verification, I opted to use a backend verification technique where the ip address is the pointer to the user's cookie. Here you can get the cookie from the request object that comes with a route initiation. We can use the top function to get the cookie from the ip, the bottom function lets us get the cookie by a username.

```
<FileSystemSession {None: 'alex', 'ISmaAKvj6k-2N_3B44cLsDRK9ob4i9pia-Lm3RrkF20': 'alex'}>
<FileSystemSession {None: 'alex', 'ISmaAKvj6k-2N_3B44cLsDRK9ob4i9pia-Lm3RrkF20': 'alex'}> /
session
COOKIE USER! alex
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /home HTTP/1.1" 200 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/css/home.css HTTP/1.1" 304 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/css/main.css HTTP/1.1" 304 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/css/navbar_animation.css HTTP/1.1" 304 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/javascript/sql.js HTTP/1.1" 304 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/css/sqlpage.css HTTP/1.1" 304 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/javascript/homepage.js HTTP/1.1" 304 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/javascript/request.js HTTP/1.1" 304 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/javascript/check_sign_ins.js HTTP/1.1" 304 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/javascript/animation.js HTTP/1.1" 304 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/javascript/create_new_account.js HTTP/1.1" 304 -
ISmaAKvj6k-2N_3B44cLsDRK9ob4i9pia-Lm3RrkF20
THE TOTAL PATH: C:\Users\alexk\OneDrive\Desktop\Websites\Languages-Database-Setup-main\flask\FlaskApp\flask\DataBases\GameData.db
PASSED
{'name': 'alex', 'failed': False, 'rank': 1, 'hits': 2894, 'medals': ('Degenerate',), 'achievements': [('Starter',), ('Loser',), ('Noob',)], 'picture': ('6',)}
127.0.0.1 - - [18/Sep/2025 22:28:13] "POST /get_profile_data HTTP/1.1" 200 -
127.0.0.1 - - [18/Sep/2025 22:28:13] "GET /static/images/profiles/6.png HTTP/1.1" 304 -
```

```
ISmaAKvj6k-2N_3B44cLsDRK9ob4i9pia-Lm3RrkF20
```

We can see the cookie printed in the console once a user signs in. The cookie is stored using flask-session.

Password Management

Since the website contains accounts with profiles, we have to make sure to keep their information safe. To make sure there are no account takeovers, we result in the use of hashing and hash comparison.

```python
def hash_data(data) -> str:
    """
    Using hash type md5 to hash passwords
    """
    hasher = hashlib.md5(data.encode())
    # hasher.update(data.encode('utf-8'))
    return hasher.hexdigest()


def compare_sign_in(username, password, message) -> tuple:
    """
    when a user attempts a sign in, we have to re-hash the inputed password
    and compare the hashes
    to the password hash under the inputed username. if no username exists or
    the hashes dont match,
    we return an error code and message describing what was incorrect.
    """
    base = SqlDatabase()
    try:
        data = base.fetchone(f"""SELECT PASSWORD, USERNAME
                                 FROM UserData WHERE USERNAME = '{username}';""")
        print('THE DATA IS:', data)
        has = hash_data(password)
        # print(data[0], has)
        if data[1] == username and data[0] == has:
            return (True, message)
        if data is None or data[0] != has:
            message = 'Wrong username or password'
            print('MESSAGE:', message)
            return (False, message)
    except Exception as e:
        print("SIGN IN ERROR:", e)
        message = f'{e}'
        print('MESSAGE:', message)
        return (False, message)

    return (False, message)
```

These two functions are what take care of password management and comparisons. When a user types in their username and password,we first check to see if that username. Then we hash the password that was imputed by the user and compare the imputed hashed password to the saved user password in the database. If the hashes match, that means the password is the same and then returns a statement telling the backend that this is the correct credentials and allows a login. The backend then saves the users ip address as a cookie pointer.

404 Game Boundary Case Testing & Data Integrity

When testing the entirety of the game and how it tracks the player, I implemented a one way connection where the frontend sends the data of the current players position and game information to the backend where the flask route functions update the SQL database.

Hits Counter

When taking note of hits, In the front end, to stop DDOS attacks to the backend, we only update the hit count after each boss stage. We can see that in this javascript function here:

```
function boss_upgrade(){
    if(boss == null){
        boss = document.getElementById('sadface');
    }


    if(tutorial_stage == 1){
        setTimeout(() => {game_paused = true; hide_show_tutorial(true, "AVOID THE BOSS!")}, 3000);
        setTimeout(() => {game_paused = false; hide_show_tutorial(false); tutorial_stage += 1}, 8000);
    }

    upgrading_boss = true;
    //boss.style.animation = 'boss_upgrade_animation 0.8s';
    boss.classList.add('boss_animation_class');
    setTimeout(() => {upgrading_boss=false;facestages();}, 900);
    send_request({'hits' : ClickCounter-before_click}, 'get_clicks', null);
    before_click = ClickCounter;

    if(tutorial_stage == 4){
        setTimeout(() => {game_paused = true; hide_show_tutorial(true, "The boss upgrades over time. It will get harder from here")}, 1000);
        setTimeout(() => {game_paused = false; hide_show_tutorial(false); tutorial_stage += 1}, 4000);
    }
}
```

```
send_request({'hits' : ClickCounter-before_click}, 'get_clicks', null);
```

We send the amount of clicks we counted by subtracting the total number of clicks from the amount of clicks we had from the previous stage.

We can see it in action here:

Account: Click_Test

Before:

	PlayerID	USERNAME	Money	Medal	Hits	Rank	PASSWORD	Wins	Picture
1	1	alex	20	11	2894	1	81dc9bdb52d04dc20036dbd8313ed055	0	6
2	2	INSERT	20	1	478	2	e10adc3949ba59abbe56e057f20f883e	0	8
3	3	22197	20	0	0	5	cf9bb12aa47ab3e8ce353b001b8b79e4	0	0
4	4	22343	20	0	0	6	91f724a7a4de4b13f12e65ce67ed0e23	0	0
5	5	22177	20	0	0	7	e10adc3949ba59abbe56e057f20f883e	0	0
6	6	22146	20	0	0	8	358aaf4a91006722f704e432b82fa60e	0	0
7	7	hi_there	20	0	152	3	e10adc3949ba59abbe56e057f20f883e	0	4
8	8	Aaronatedih	20	0	0	9	cb9fbb82698d4d941edb1db8c6796fe7	0	1
9	9	new_account	20	0	0	10	e10adc3949ba59abbe56e057f20f883e	0	5
10	10	22046	20	0	0	11	e10adc3949ba59abbe56e057f20f883e	0	3
11	11	test	20	0	0	12	5b4e332e96806d58418f73d16c6a7227	0	1
12	12	llovr[]SELECTA	20	0	55	4	1c63129ae9db9c60c3e8aa94d3e00495	0	8
13	13	Click_Test	20	0	0	1000	e10adc3949ba59abbe56e057f20f883e	0	1

13	13	Click_Test	20	0	0	1000	e10adc3949ba59abbe56e057f20f883e	0	1

After a 10 clicks in the game:

	PlaverID	USERNAME	Monev	Medal	Hits	Rank	PASSWORD	Wins	Picture
1	1	alex	20	11	2894	1	81dc9bdb52d04dc20036dbd8313ed055	0	6
2	2	INSERT	20	1	478	2	e10adc3949ba59abbe56e057f20f883e	0	8
3	3	22197	20	0	0	6	cf9bb12aa47ab3e8ce353b001b8b79e4	0	0
4	4	22343	20	0	0	7	91f724a7a4de4b13f12e65ce67ed0e23	0	0
5	5	22177	20	0	0	8	e10adc3949ba59abbe56e057f20f883e	0	0
6	6	22146	20	0	0	9	358aaf4a91006722f704e432b82fa60e	0	0
7	7	hi_there	20	0	152	3	e10adc3949ba59abbe56e057f20f883e	0	4
8	8	Aaronatedih	20	0	0	10	cb9fbb82698d4d941edb1db8c6796fe7	0	1
9	9	new_account	20	0	0	11	e10adc3949ba59abbe56e057f20f883e	0	5
10	10	22046	20	0	0	12	e10adc3949ba59abbe56e057f20f883e	0	3
11	11	test	20	0	0	13	5b4e332e96806d58418f73d16c6a7227	0	1
12	12	Ilovr[]SELECTA	20	0	55	4	1c63129ae9db9c60c3e8aa94d3e00495	0	8
13	13	Click_Test	20	0	10	5	e10adc3949ba59abbe56e057f20f883e	0	1

13	13	Click_Test	20	0	10	5	e10adc3949ba59abbe56e057f20f883e	0	1

We can see that yes, it updates to 10 clicks which also in fact updates the leaderboard where the account now sits at 5th place which is correct.

Rank	USERNAME	Hits
1	alex	2894
2	INSERT	478
3	hi_there	152
4	Ilovr[]SELECTA	55
5	Click_Test	10
6	22197	0
7	22343	0
8	22177	0
9	22146	0

The function of code that sorts this is:

```python
def update_leaderboard() -> None:
    """
    this function sorts the ranks of the users by how many hits
    they have. The more hits they have, the higher their rank is

    user with the highest rank gets the goat achievement
    """
    names = {}
    base = SqlDatabase()
    data = base.fetchall("SELECT USERNAME, hits FROM UserData", close=True)
    print(data)
    for item in data:
        name = item[0]
        hits = item[1]
        names[name] = hits

    sorted_items = sorted(names.items(), key=lambda item: item[1],
                          reverse=True)

    sorted_dict = dict(sorted_items)
    print(sorted_dict)

    for rank, name in enumerate(sorted_dict.keys()):
        print(name, rank + 1)
        update_sql('UserData', 'rank', rank + 1, 'USERNAME', name)
```

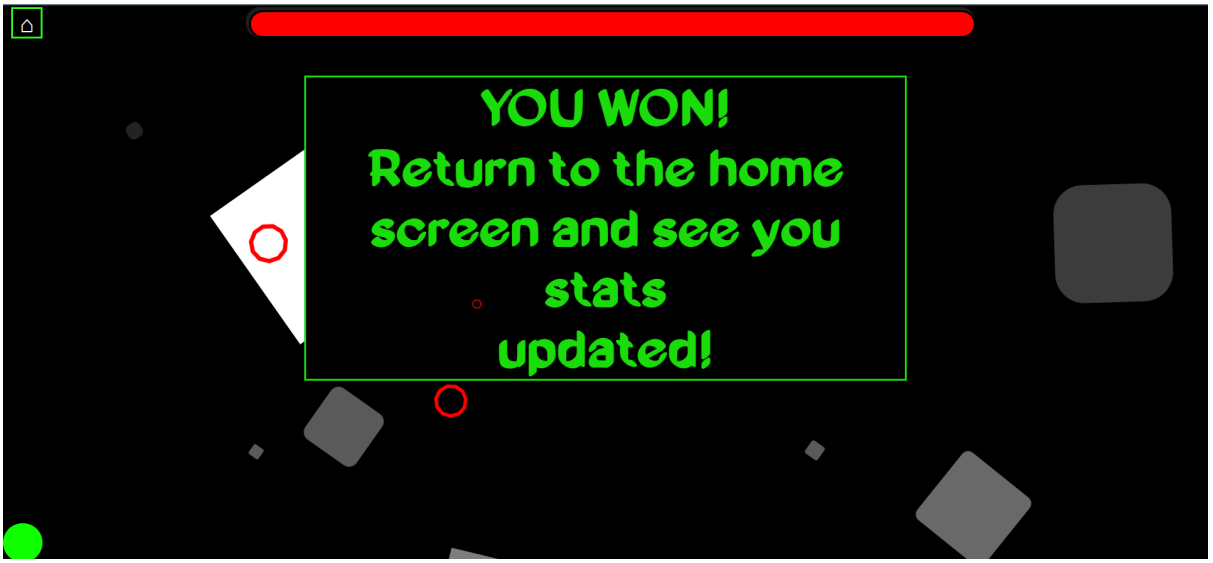
And it sorts every time the filter button is clicked to make sure that the information is fresh.

Winning Game

On the front end, we have to detect when the player has one. This is done by a javascript function where we check the next stage the boss evolves too and if the stage is to the ‘dead’ stage, we tell the game to stop and tell the player that the game is done.

```javascript
    if(stage == 'dead'){
        boss_dead = true;
        let el = document.getElementById("WIN_TEXT");
        el.style.visibility = 'visible'
        send_request("dataata", 'record_win');
    }
```

The ‘boss_dead’ variable is used in update loops for the game and pauses them so the game doesn't continue on.



We then send a request to the backend to tell flask that a user has completed the game and to record the win.

```python
@app.route("/record_win", methods=['POST', 'GET'])
def record_win():
    user = None
    print('RECORDING WIN!')
    try:
        # get user cookie
        user = get_user_by_cookie(request, session)
        print("USER:", user, get_cookie(request))
        database = SqlDatabase()
        medal = database.fetchone(f'''Select Medal FROM UserData
                                  where USERNAME = "{user}"''')
        medal = int(medal[0])
        # check what medal the user has
        # after each win, the medal value increases
        if medal < 11:  # see if at max medal level
            medal += 1
        update_sql("UserData", "Medal", medal, "USERNAME", user)
        print("UPDATED SQL MEDAL DATA!")

        database = SqlDatabase()
        id = database.fetchone(f'''Select UserData.PlayerID From UserData Where
                           USERNAME = "{user}";
                           ''', close=False)
        hasa = database.fetchone(f'''Select PlayerID From Awarded Where
                           PlayerID = {id[0]} AND AwardID = 2;
                           ''', close=True)
        print("HE DOES HAVE:", hasa, id)
        if hasa is None:
            querrey = f"""
                INSERT INTO Awarded
                (AwardID, PlayerID)
                VALUES
                (2 , {id[0]})
                    """
            database = SqlDatabase()
            database.execute(querrey, close=False)
            database.connection.commit()
    except Exception as e:
        # if user isnt signed in, ignore
        print(e)
        return jsonify({})
    return jsonify({})
```

We can see in the console that we have indeed gotten a win

```
127.0.0.1 - - [18/Sep/2025 18:44:20] "GET /static/images/faces/9.png HTTP/1.1" 200 -
127.0.0.1 - - [18/Sep/2025 18:44:23] "GET /static/images/faces/10.png HTTP/1.1" 404 -
127.0.0.1 - - [18/Sep/2025 18:44:23] "POST /get_clicks HTTP/1.1" 200 -
127.0.0.1 - - [18/Sep/2025 18:44:24] "GET /static/images/faces/10.png HTTP/1.1" 404 -
RECORDING WIN!
'ISmaAKvj6k-2N_3B44cLsDRK9ob4i9pia-Lm3RrkF20'
127.0.0.1 - - [18/Sep/2025 18:44:24] "POST /record_win HTTP/1.1" 200 -
127.0.0.1 - - [18/Sep/2025 18:44:24] "GET /static/images/faces/10.png HTTP/1.1" 404 -
```

```
RECORDING WIN!
```

Since the account is already level the medal ID value stayed the same:

	PlaverID	USERNAME	Monev	Medal	Hits	Rank	PASSWORD	Wins	Picture
1	1	alex	20	11	2894	1	81dc9bdb52d04dc20036dbd8313ed055	0	6

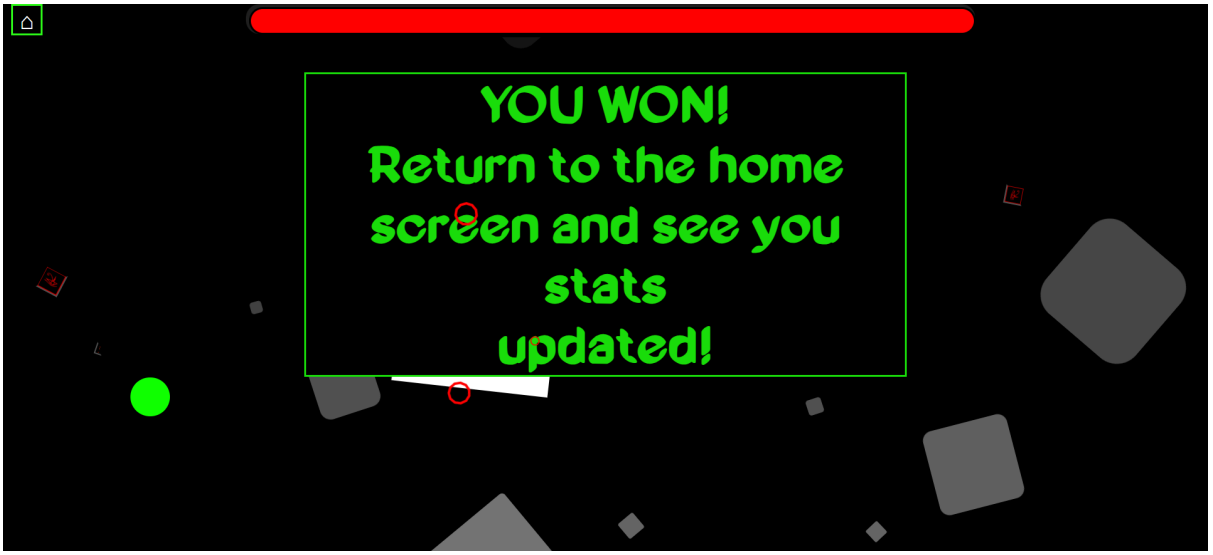
On a different account (‘INSERT’ is the account name I am going to use):

Before winning:

2	2	INSERT	20	0	328	2	e10adc3949ba59abbe56e057f20f883e	0	8

0 (Medal ID Value)

After Winning:



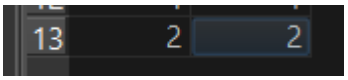

```
RECORDING WIN!
USER: INSERT ISm                                    ppia-Lm3RrkF20
THE TOTAL PATH:  Open file in editor (ctrl + click)  sktop\Websites\Languages-Database-Setup-main\flask\FlaskApp\flask\DataBases\GameData.db
THE TOTAL PATH: C:\Users\alexk\OneDrive\Desktop\Websites\Languages-Database-Setup-main\flask\FlaskApp\flask\DataBases\GameData.db
127.0.0.1 - - [18/Sep/2025 19:10:51] "GET /static/images/faces/10.png HTTP/1.1" 404 -
UPDATED SQL MEDAL DATA!
127.0.0.1 - - [18/Sep/2025 19:10:51] "POST /record_win HTTP/1.1" 200 -
```

2	2	INSERT	20	1	478	2	e10adc3949ba59abbe56e057f20f883e	0	8

1

We can see that the medal has been updated from zero to one which also does not mean they get a new achievement which is the winner achievement.

	AwardID	PlayerID
1	0	1
2	4	2
3	4	3
4	4	4
5	1	1
6	4	5
7	4	6
8	4	7
9	1	7
10	4	8
11	4	9
12	4	1
13	2	2



Losing Game

Losing the game is similar to winning the game where the front end sends a post request letting the backend know that the user has lost the game. The purpose of doing this to give the user an achievement where they lose their first game.

Front End Function:

```
function take_damage(damage = 0){
    if(inv == true){
        return;
    }
    console.log("PLAYER TOOK DAMAGE!");
    player_hp += damage;
    if(player_hp <= 0){
        player.style.backgroundColor = `rgb(0, 0, 0, 0)`;
        document.getElementById('DIED_TEXT').style.visibility = 'visible';
        mouse_player.style.borderBottomColor = `rgb(0, 0, 0, 0)`;
        dead = true;
        player_dead = true;
        send_request('player_dead', 'record_death');
        return;
    }

    var hp = (player_hp / player_max_hp) * 255;
    var counter = 255 - hp;
    console.log(hp);
    player.style.backgroundColor = `rgb(${counter}, ${Math.trunc(hp)}, 0)`;
    mouse_player.style.borderBottomColor = `rgb(${counter}, ${Math.trunc(hp)}, 0)`;
}
```

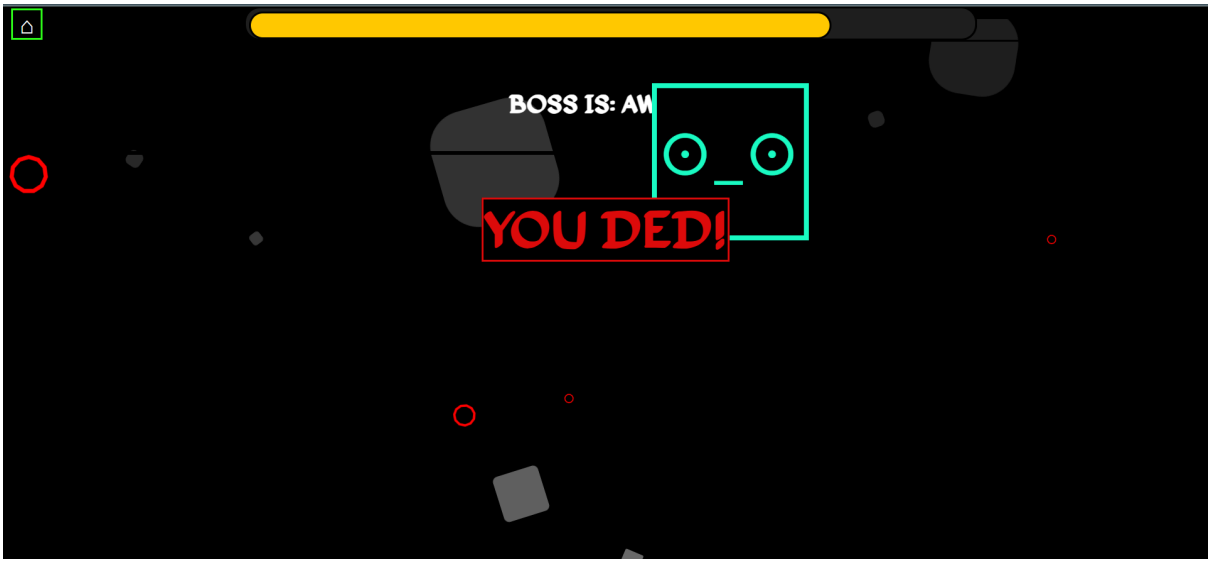
```
if(player_hp <= 0){
    player.style.backgroundColor = `rgb(0, 0, 0, 0)`;
    document.getElementById('DIED_TEXT').style.visibility = 'visible';
    mouse_player.style.borderBottomColor = `rgb(0, 0, 0, 0)`;
    dead = true;
    player_dead = true;
    send_request('player_dead', 'record_death');
    return;
}
```

Back End Function:

```
@app.route("/record_death", methods=['POST', 'GET'])
def record_death():
    # data = jsonify_request(request.get_data())
    user = None
    try:
        user = get_user_by_cookie(request, session)
        database = SqlDatabase()
        id = database.fetchone(f'''Select UserData.PlayerID From UserData Where
                               USERNAME = "{user}";
                               ''', close=False)
        hasa = database.fetchone(f'''Select PlayerID From Awarded Where
                               PlayerID = {id[0]} AND AwardID = 1;
                               ''', close=True)
        print("HE DOES HAVE:", hasa, id)
        if hasa is None:
            querrey = f"""
                INSERT INTO Awarded
                (AwardID, PlayerID)
                VALUES
                (1 , {id[0]})
                    """
            database = SqlDatabase()
            database.execute(querrey, close=False)
            database.connection.commit()

    except Exception as e:
        print(e)
        return jsonify({})
    return jsonify({})
```

Dead Screen:



Testing on the same account (‘Click_Test’):
Before:

	AwardID	PlayerID
1	0	1
2	4	2
3	4	3
4	4	4
5	1	1
6	4	5
7	4	6
8	4	7
9	1	7
10	4	8
11	4	9
12	4	1
13	2	2

	PlaverID	USERNAME	Monev	Medal	Hits	Rank	PASSWORD	Wins	Picture
1	1	alex	20	11	2894	1	81dc9bdb52d04dc20036dbd8313ed055	0	6
2	2	INSERT	20	1	478	2	e10adc3949ba59abbe56e057f20f883e	0	8
3	3	22197	20	0	0	6	cf9bb12aa47ab3e8ce353b001b8b79e4	0	0
4	4	22343	20	0	0	7	91f724a7a4de4b13f12e65ce67ed0e23	0	0
5	5	22177	20	0	0	8	e10adc3949ba59abbe56e057f20f883e	0	0
6	6	22146	20	0	0	9	358aaf4a91006722f704e432b82fa60e	0	0
7	7	hi_there	20	0	152	3	e10adc3949ba59abbe56e057f20f883e	0	4
8	8	Aaronatedih	20	0	0	10	cb9fbb82698d4d941edb1db8c6796fe7	0	1
9	9	new_account	20	0	0	11	e10adc3949ba59abbe56e057f20f883e	0	5
10	10	22046	20	0	0	12	e10adc3949ba59abbe56e057f20f883e	0	3
11	11	test	20	0	0	13	5b4e332e96806d58418f73d16c6a7227	0	1
12	12	llovr[]SELECTA	20	0	55	4	1c63129ae9db9c60c3e8aa94d3e00495	0	8
13	13	Click_Test	20	0	10	5	e10adc3949ba59abbe56e057f20f883e	0	1

13	13	Click_Test	20	0	10	5	e10adc3949ba59abbe56e057f20f883e	0	1

After:

	AwardID	PlaverID
1	0	1
2	4	2
3	4	3
4	4	4
5	1	1
6	4	5
7	4	6
8	4	7
9	1	7
10	4	8
11	4	9
12	4	1
13	2	2
14	4	13
15	1	13

14	4	13
15	1	13

The Top column is when the user first plays the game
The Second column was when the game was lost.

Website Iterations

First Iteration

Summery

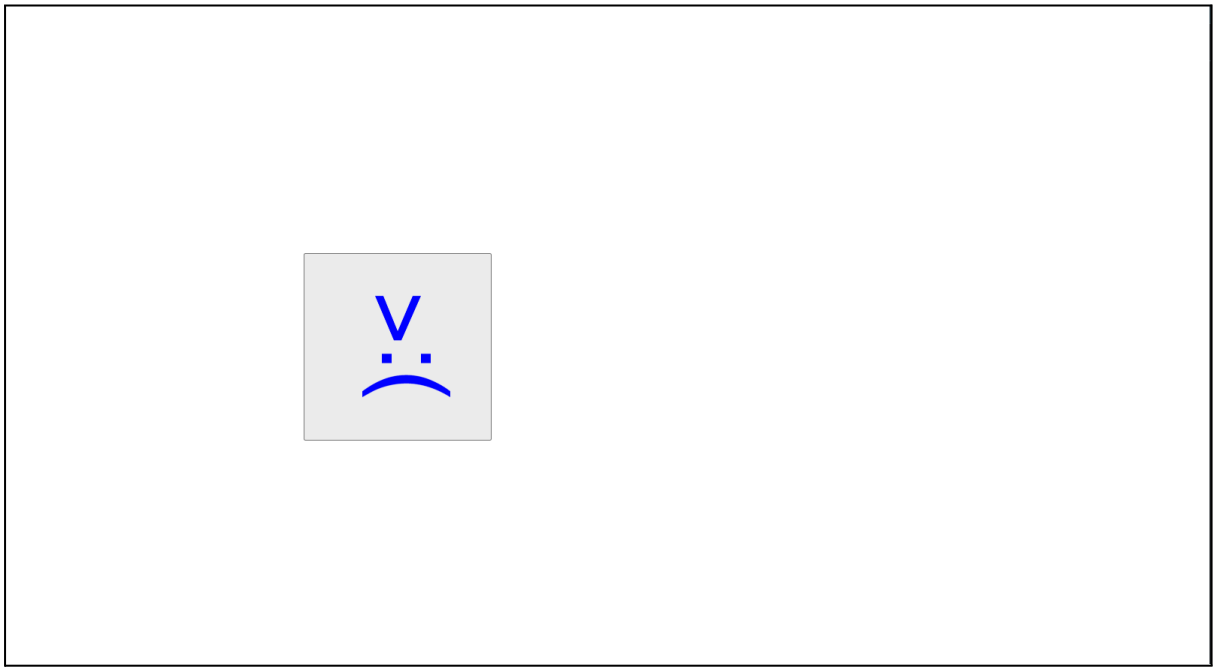
During the early age of website development, the goal was to make your website stand out as much as possible. If we look at early website pages (like the old lings car website), we can see a lot of flashing colors and moving decorations, all to keep the user hooked to the page and interested. Nowadays, developers try to make their application as widespread and usable as possible. This means keeping your website to the most minimalist and simplest layout so that anyone can use it with ease. I have also gone for this approach by keeping the website clean and out of random gunk that isn't a necessity to the user.

At the top Navigation bar, I have the user options. When the user first enters the website, they will be asked to sing in, or make an account. Once they click the sign in button, the user will get a pop up window to put in their username or password. If they click the create new account button, the user will get a different pop up where they can choose their username and password where they will confirm the password twice. If I have time in the future, I will create a validation form where they have to confirm their email address to continue the account creation.

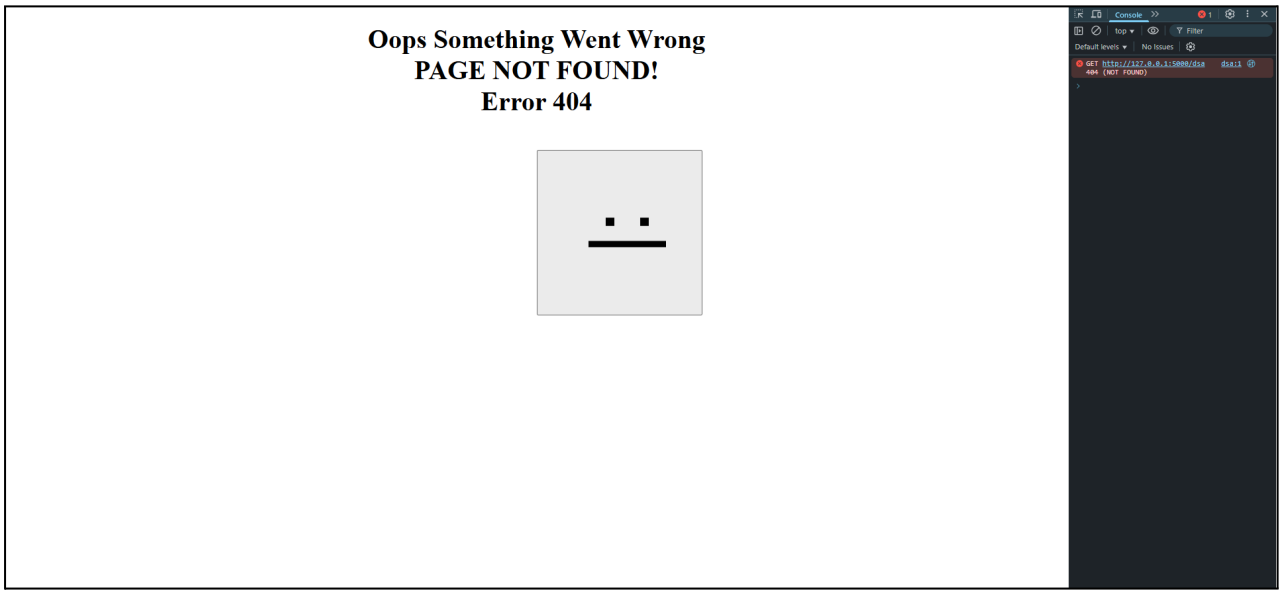
The stats button found at the top of the page, is where the user can see their own statistics, which is directly linked to their spot in the Sql Database. (only once they have signed in can they view their stats)

The body of the page will contain the play button (where they can actually play the game) and a leaderboard. The leaderboard will be a direct representation of the Sql Database. Here, the user can search for certain stats like how many players have the gold medal, or what players have a certain achievement.

Development Log

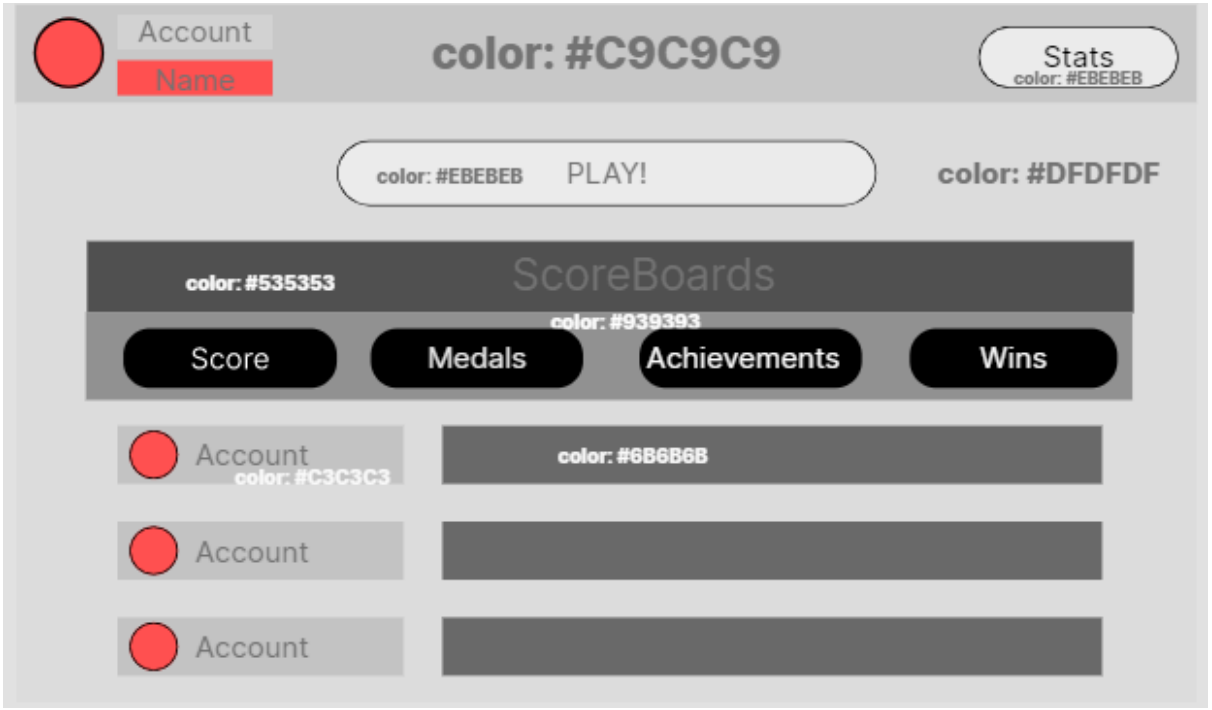


To start off with the website, I started with a simple button and flask route that routed all unknown routes (404) to one page.



The next step in development was to start the homepage where I planned to put the user profile picture in the top left in the blue circle, with the sign in page and stats button separated. The Idea of the theme is to make a gray simple minimalist color scaled website.

Started development of the game by bouncing a squat button around the screen where the more you clicked him, the higher the click counter went.



Feedback

Name	Position	Positives	Negatives
Alex Yao	Programmer	Very simple colors and very minimalistic	It looks very ugly. Buttons are all over the place and I don't

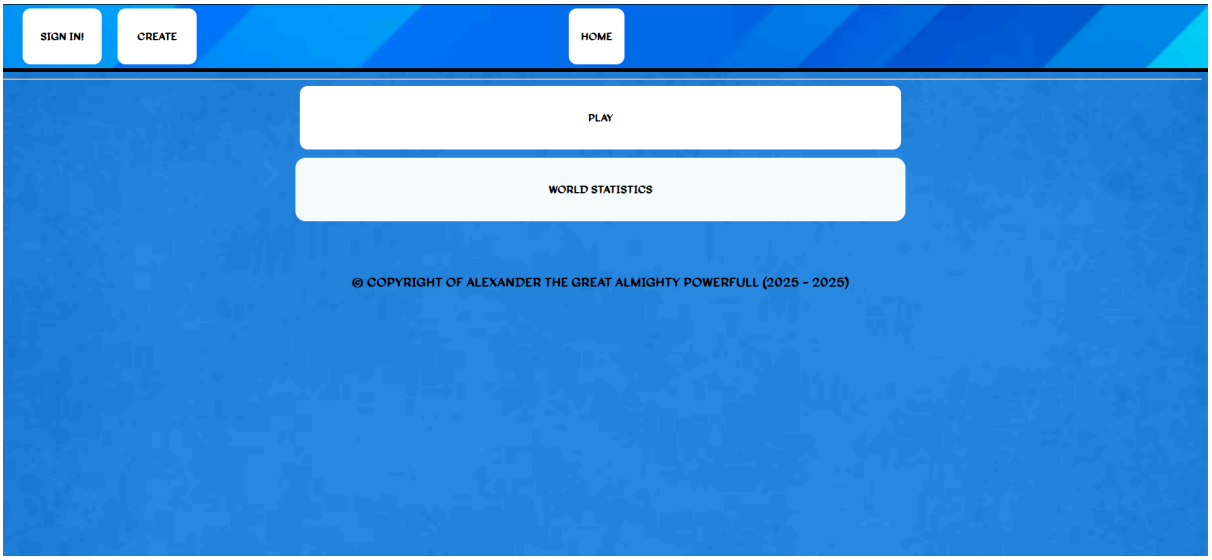
			understand what's going on with the website's layout.

Second Iteration

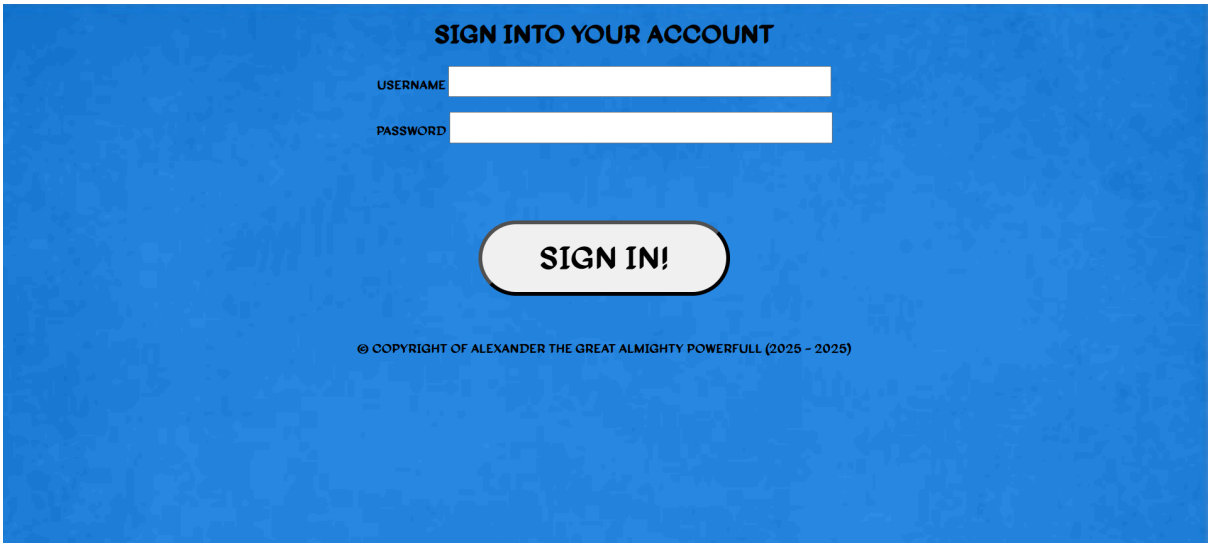
Summery

The second iteration is where I started to build up the website's foundations like the top bar, sign up / sign in pages and the main layout. Here I decide to follow common conventions with a top navigation bar that has a simple color design and the buttons are easily distinguishable.

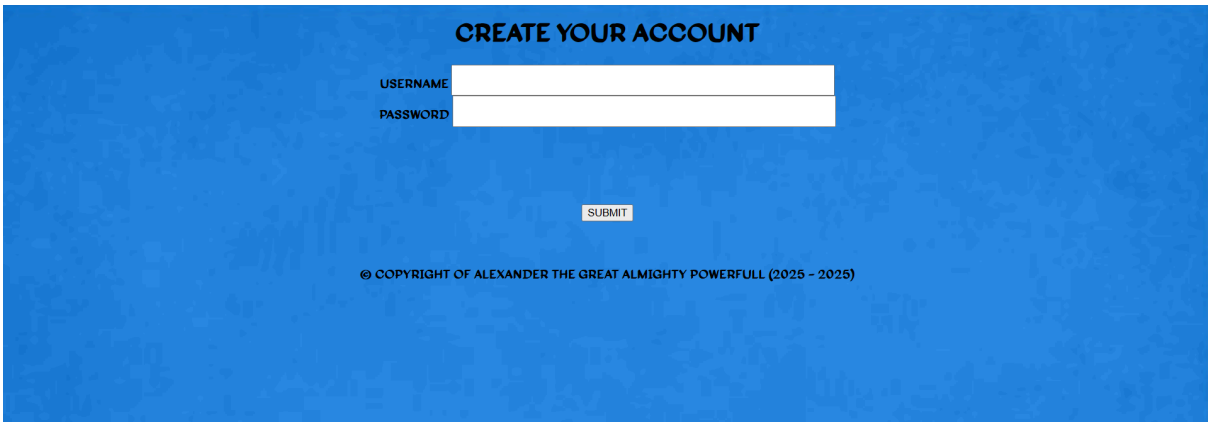
Development Log



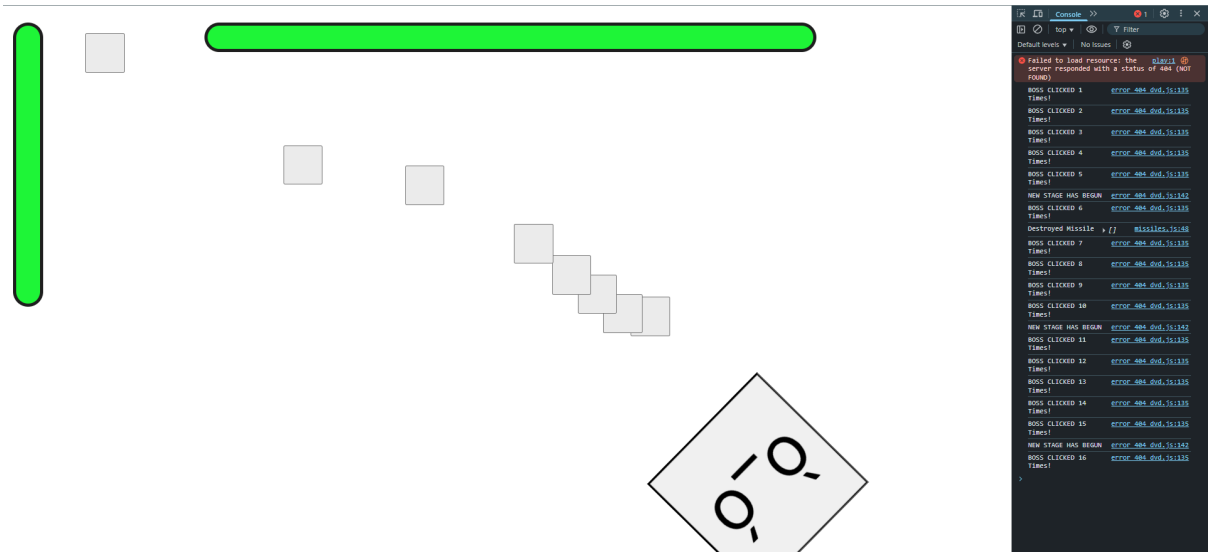
The gray website looked too bleak and boring and ugly so I started to experiment with different color pallets and themes. In this attempt, I tried to go for a cold blue color scheme.



Started developing the sign in page.



Started creating the create account page.



Started adding health bars to the boss and experimenting with move movement patterns.

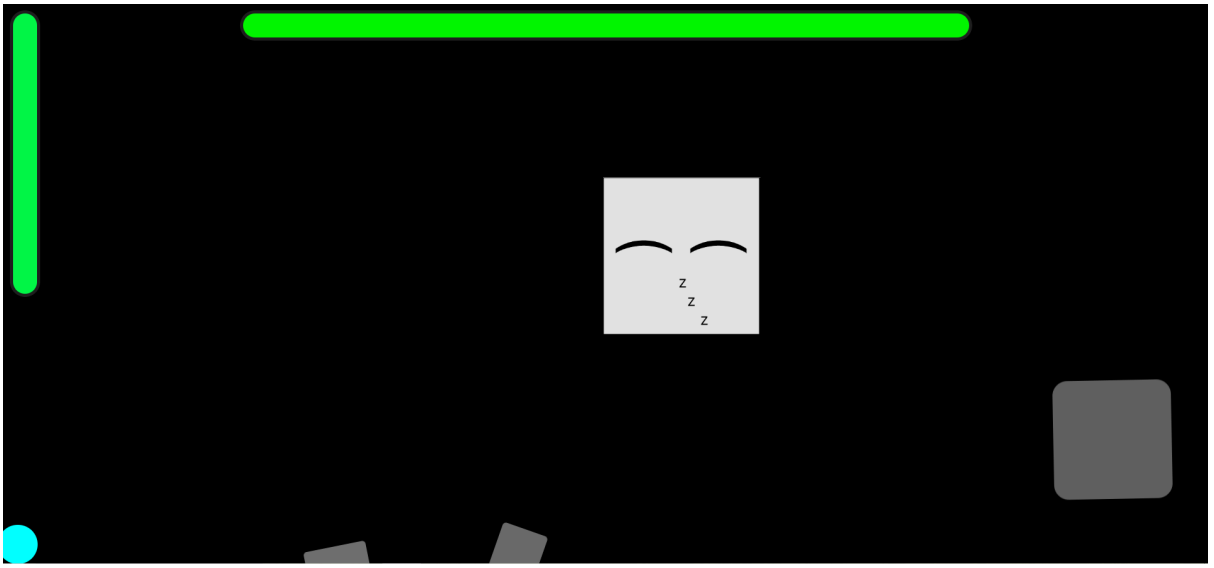
Third Iteration

Summery

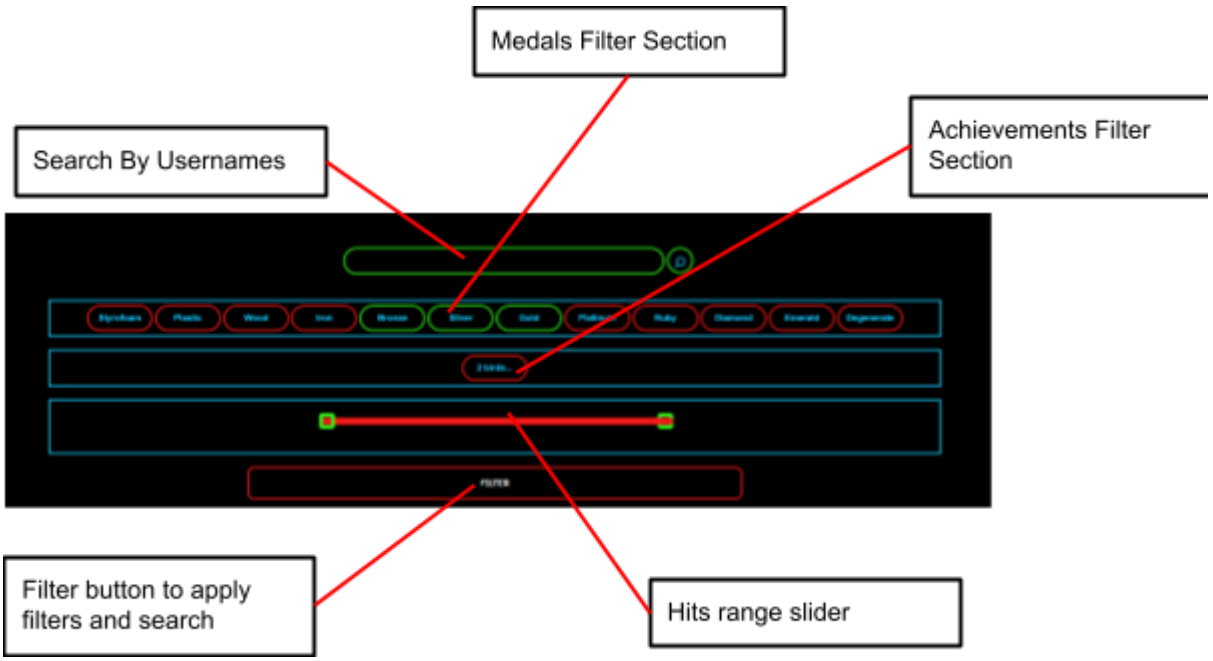
The third iteration is where I start to finalise the themes, functionality and usability. I settled on a neon theme. Most functions on the website like the filtering system was done and working.

Development Log

Changed the entire website theme in css to be consistent.



For the game, I decided to make the background dynamic and dark so that the game objects stand out more and are easier to distinguish.



Final Iteration

Summery

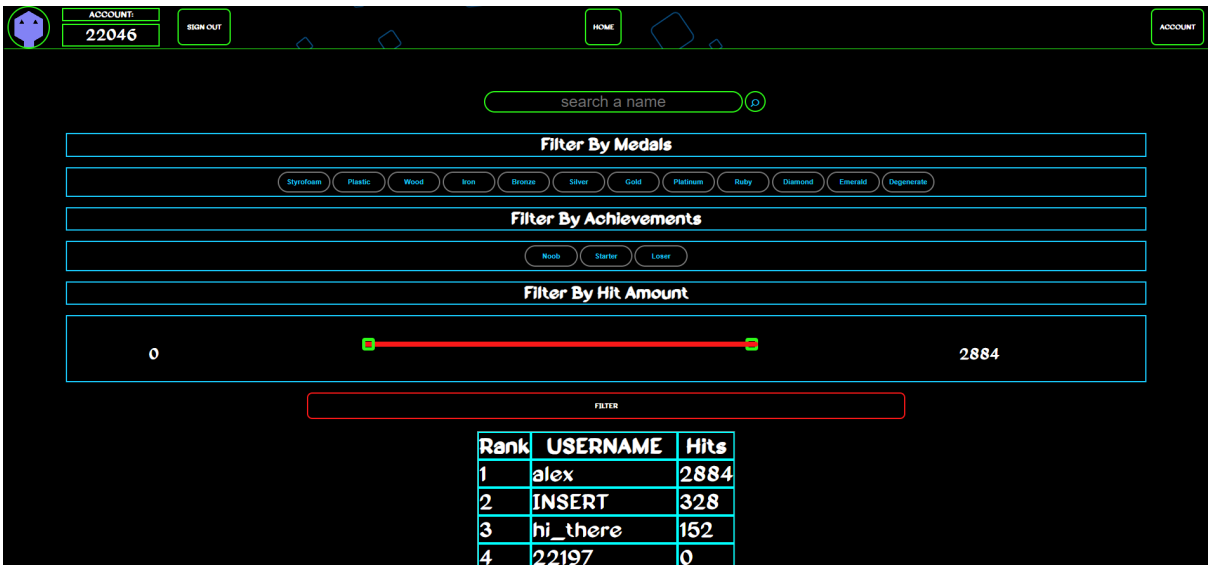
The final iteration was mostly cleaning up the website, giving confusing things like labels to the filtering options and a tutorial to the game. It also came with some bug fixes like the many to many tables not working in the filtering system due to a foreign key mismatch.

Development Log

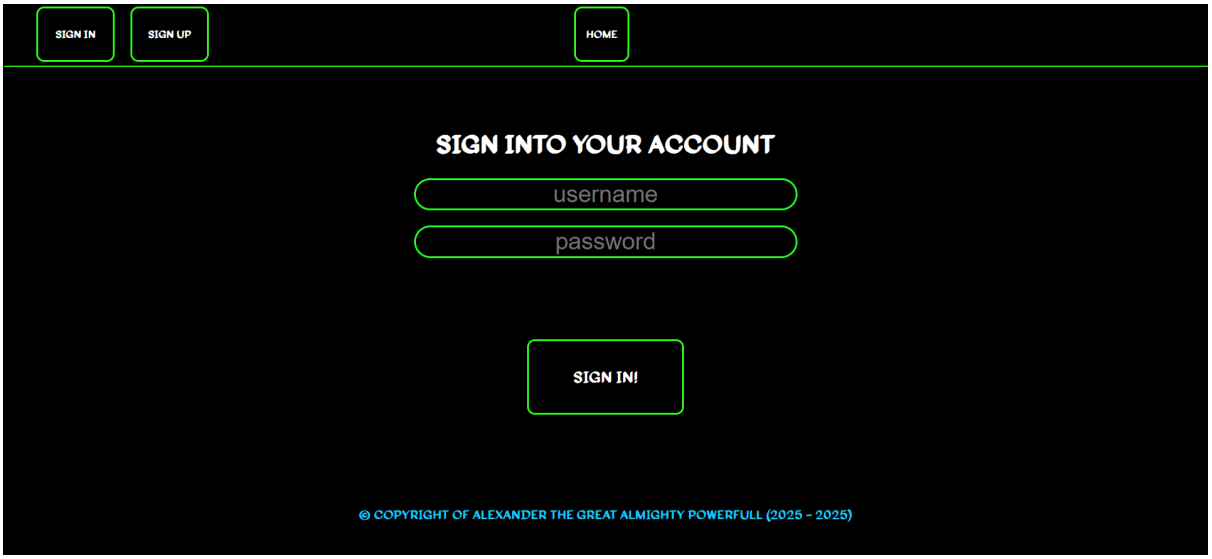
Added a tutorial that appears and can be optionally skipped with the top right button that says “skip tutorial” and added a home button as before you could only go back to the home page by retypering ‘/home’ or pressing the back button.

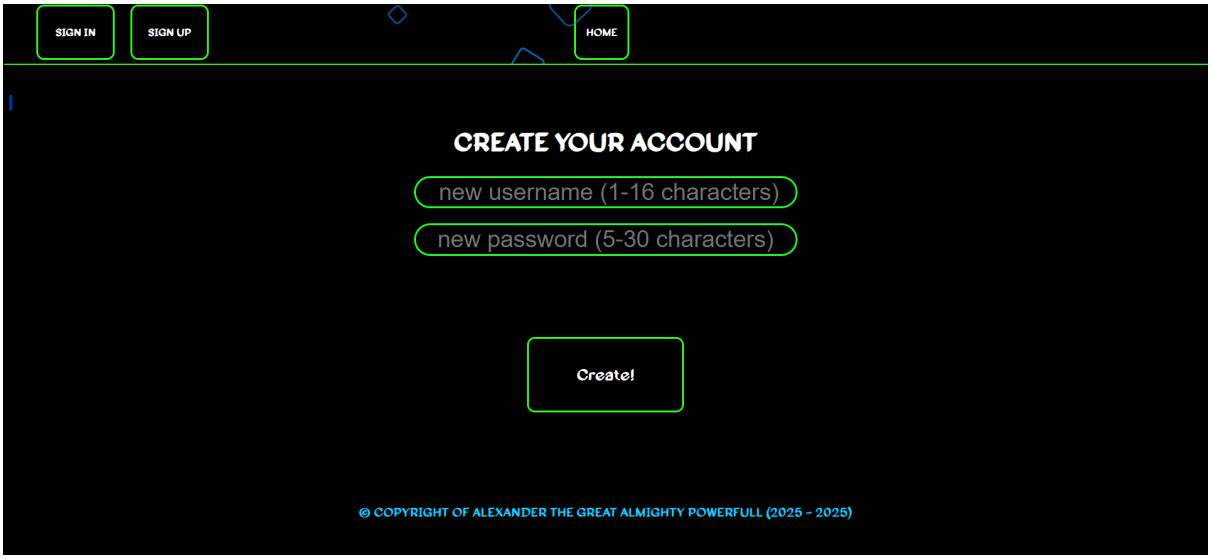
After some website theme changes, I have decided to finalise the theme to a neon color scheme. To follow the heuristic of minimalism, I decide that single bright colors with simple shapes are the best option to keep with the minimalist feel.

Added a new about button that leads you to the about page.

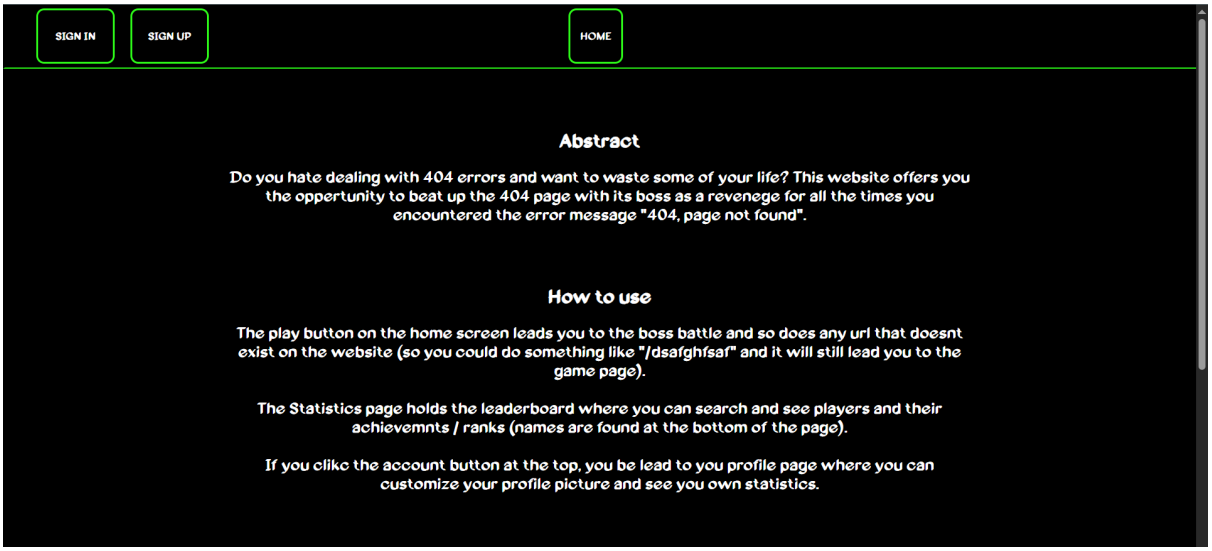


Labeled what the filters do





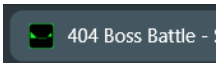
The sign in page and account creation page did not change except for the text in the create account inputs to tell the user beforehand how long the inputs can be.

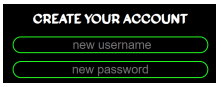
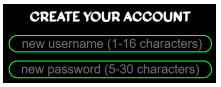


New page that goes more in depth of what the site is (some lore too).

Website Feedback

Name	Section of Website	Negatives	Positives	Fixes / Solutions
Alex Yao (programmer)	All of it	The game was slightly confusing for new players	Really good aesthetics and stats page	Add a tutorial to the game.
Riki Smillie (programmer)	All of it	• Really confusing (especially game)	Animations on the buttons were really cool The theme was also cool	Only shows the players data in world statistics once it is searched up so on page load, the website will automatically load every user onto a table.
Alex Yao (programmer)	Game	• It's way too hard. I lose by the third level	• The game is fun.	Decrease difficulty of the game so that it's more playable and enjoyable.
Andrew Wu (programmer)	All of it except the tutorial	If a user opts to skip the tutorial, they will not understand how to play the	I really like the animations. It makes the website feel “alive”.	Your website should not be called “DataBase Site - Home”

		game. Personally, I think the game should be more intuitive and if the user decides to skip the tutorial, there should still be more guidance on the main game page. This would account for the fact that some users may expect a system to be intuitive and would always skip any tutorials because they, in their ultimate conceit, believe that they know how to operate all technology.		Changed the title. 
Marshall (non-programmer)	Whole website	The game is not replayable. It gets boring after a while	The UI is very easy to navigate.	
Charlotte (Non-programmer)	Whole Website	Confused by the game. Bad trackpad on testing laptop	Everything else is good.	Made the tutorial more intuitive by stopping the game and telling the user what to do in the first boss stage.
George (non-programmer)	Whole Website	The account creation took a while because I had to keep re-entering my password, and I didn't know what my username needed until it	I thought the game was fun, and the filter seemed very cool.	For the usernames, I will tell the user beforehand what the requirements are. Before:

		told me one at a time.		 After: 

Website Relevant Conventions

When developing the website, there were many techniques that were used to improve and create the design of the pages. I used css, html and javascript to give life to the pages. It also links back to heuristics.

Front End Javascript & Routing

The front end of the page has to have some way of communicating data to the backend. The simple html approach of routing is not enough to complete the required functions for the website to work properly. This introduces the javascript language which is widely supported for frontend programming. The communication protocol that I found was best to transmit data was Google's Ajax where Json data could be sent from javascript to python. This is a common library used throughout many websites in the world and is a common standard.

```
function send_request(command = '', func = '', rfunc = null, bfunc = null) {
    var value = command;
    const url = `/${func}`;  //http://127.0.0.1:5000
    var mode = 'POST';
    var data = {'data' : value};

      $.ajax({
            url: url,
            method: mode,
            dataType : 'json',
            data: JSON.stringify(data),
            crossDomain: false,
            mode : 'no-cors',
            dataType : 'json',
          }).done(function (data) {
            try{
              if(rfunc != null){
              rfunc(data);
              }
            }catch(e){
              console.assert(e);
            }

          }).fail(function (error){
            alert(error);
          });

  }
```

For handling send requests, I made a single function to handle the front end cleanly. THe function first takes the message of what the user wants to send. It will then jsonify it so that the format stays the same every time it's sent. The second argument is what route you want to send the data to. The third argument is for the return once the backend has received data. The backend will then send a return with data that will be sent to the imputed function. The last argument is going to be an error argument where if the request fails, it will send the error to the imputed function.

```
function make_data_table(data){
  if(data['failed'] == true){
    console.log("OH NO! I Can't seem to find what you're looking for!");
    return;
  }

  var table = document.getElementById('data_table');
  var table_data = data;

  var tables = {};
  var table_data_holder = [];
  var data_string = '<tr>';
  for(var column in data['columns']){
    data_string += '<th>' + data['columns'][column] + '</th>';
    tables[data['columns'][column]] = [];
    //console.log(data['columns'][column]);
  }
  data_string += '</tr>';
  //console.log(data['data']);

  for(var item in data['rows']){
    data_string += '<tr>';
    for(value in data['rows'][item]){
      data_string += "<td>" + data['rows'][item][value] +"</td>";
      //console.log(data['rows'][item][value]);
    }
    data_string += '</tr>';
  }

  data_string = data_string + '</tr>'
  //console.log(data_string);

  table.innerHTML = data_string;

  make_buttons(data);

}
```

When receiving data from the database, we have to convert it to a visual representation. This is where table creation came in. I made a function that creates a table and displays the data. It first finds the table object in html, then uses the returned data from a data request like the rows and columns and structures them to a readable format for a human visually.

CSS Conventions

CSS was used to make animations and keep the website consistent. After each iteration I would improve the CSS and in turn improve the website. By focusing on animation and adding small aesthetic parts to the website, it can make the website look cleaner and feel more enjoyable to use.

Variable Conventions

To make CSS faster and easier to write, I added a base root class that holds variables like colors and border width sizes that can be reused on any other CSS files.

```css
:root {
    --button1: #5fb9e67c;
    --button1_hover: rgb(138, 222, 239);
    --cream: #e8e8e8;
    --bluez: #0077b6;
    --darkblue: #000000;
    --animation_button_time: 0.3s;
    --border_color: rgb(49, 251, 27);
    --border_red_color: rgb(251, 27, 27);
    --border_gray_color: rgb(122, 122, 122);
    --border_width: 3px;
    --button_bg_color: black;
    --button_text_color: white;
    --button_hover_color: black;
    --button_hover_border_color: rgb(27, 202, 251);
}
```

To access a variable and use the value assigned to it, we can use:

```css
var(--border_color);
```

To access the var we use the var() function with two dashes and then followed by the variable name.

CSS Convention examples

The gray squats in the background is an animated background using CSS where the cubes move upwards and fade away. I thought that giving the website some dynamic backgrounds can make it stand out and give some more life to the webpage.

I added a similar effect when making the top navigation bar so when the user hovers over the top bar with their mouse, the top bar lights up with the animation.

Before hovering over a button:

After hovering over a button:

I thought that adding button animations that widen and change color, makes it clear of what the user is going to click on. It also adds to the idea that the website is for gaming as in most games, buttons like the play buttons are animated. Almost all big buttons on the website have this animation.

The code for the CSS animation (for the play button):

```css
.play{
    width: 50%;
    height: 80px;
    border-radius: 10px;
    font-family: 'cool';
    background-color: var(--button1);
    transition: 0.3s;
    border-width: 0px;

    border-width: var(--border_width);
    border-color: var(--border_color);
    border-style: solid;
    background-color: var(--button_bg_color);
    color: var(--button_text_color);
}

.play:hover {
    width: 60%;
    background-color: var(--button_hover_color);
    border-color: var(--button_hover_border_color);
    font-size: larger;
    border-radius: 50px;
}
```

HTML Conventions

For HTML conventions, I followed some basic heuristics for website design. This meant making the website as minimalistic as possible to not confuse users and make them have the best experience possible. HTML conventions mostly consist of following basic rules (aka the heuristics) and marking the website as user friendly to experienced and non experienced web surfers. I did this by having a standing out color palette by outlining important parts of the website in neon colors to stand out in the dark backgrounds, and position HTML elements in calculated positions where they feel the most comfortable for a user.

Top Navigation Bar

When a user is not signed into an account, the webpage will display this navigation bar version:

The navigation bar in this state is at its simplest form where there are only three buttons to choose from. The Sign in button, Sign up button and the home page button. This simple navigation bar with its clearly labeled buttons that relate to their function, make it easy for a user to just look and understand what their purpose is.

Once a user has signed in they will see this version of the navigation bar:

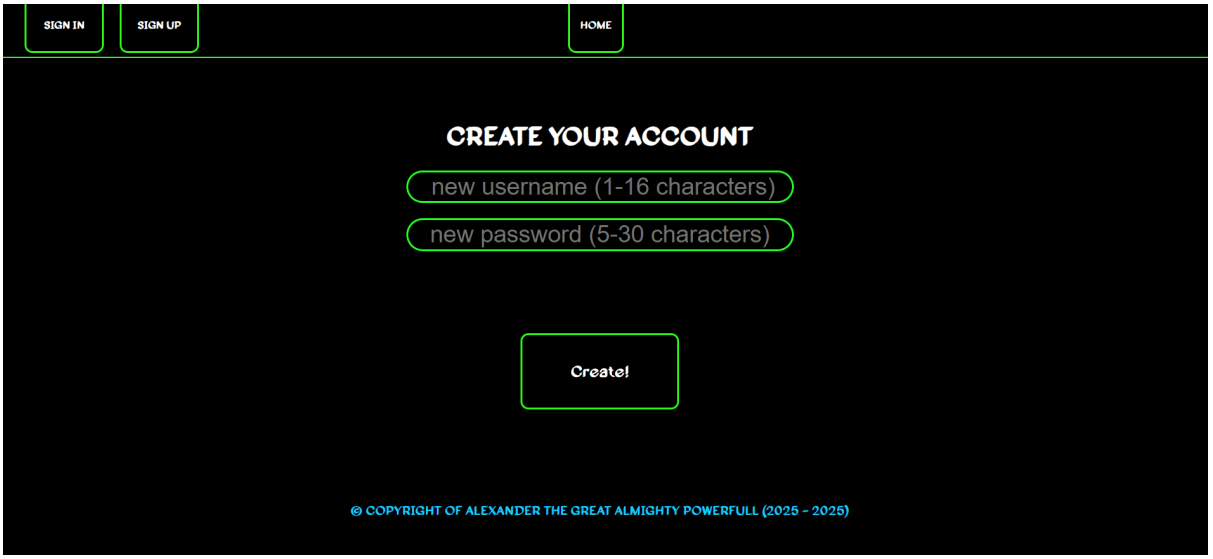
This bar becomes filled with a bit more information with a profile picture to the very left, a sign out button next to their username and a new ‘Account’ button that will lead them to their account information. I made sure to separate the sign out button away from other buttons on the navigation bar to reduce human error in case the user may misclick the button and sign out when they were in fact trying to click another button.

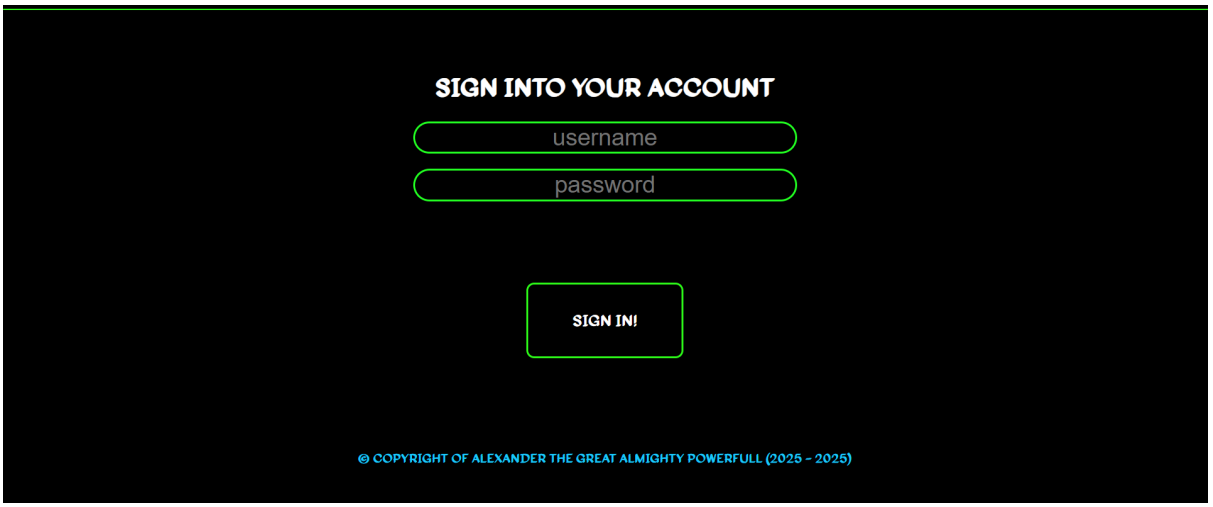
Home Page

The home page is what the user will first see when they enter the website.

When creating the home page, I made sure to remove any unneeded clutter and provide a page that only links to other pages that have their own purpose. We can see that there are three main buttons. The play button which is first because it's the main attraction of the website, the World Statistics button which is where you can interact with the database with the filtering page, and the about page, which is really the least important part of the website as it only talks about what the website is and has a bit more detail on how the game works (as the game has its own tutorial).

Sign up / Sign In Page





These pages are kept in their simplest forms with the input fields telling the user what has to be inputted into them and having the big confirmation button (either ‘create’ for the sign up, and ‘sign in’ for the sign in page) located just below. As we can see from the two pages, I made sure to keep the input fields and buttons positioned and looking consistent so users don't get confused.

Website Relevant Implications

Aesthetics

Aesthetics in terms of website design, refers to keeping the website looking good and consistent while still following the rules of Nielsen's heuristics. This means making important buttons bigger and less important ones smaller. It also just means to keep the website clutter free and make a user not think too hard when surfing the website.

When considering Aesthetics in my digital outcome, I went through many different themes and looks for the website through my iterations. To make my website look pleasing to the eye, I decided to make the website look as simple as possible. This led me to come up with a neon design that outlines buttons and objects in a bright neon color that are important , and has a black simple background.

End-User Considerations

End-User Considerations in terms of a website outcome, means that the website should be simple, easy to navigate and not confusing for newcomers. This can mean making all search bars look the same so no matter what page you are on, the user will always know what the search bar looks like, or following website heuristics about giving control to the user and matching things to the real world.

When creating the website I implemented and built the website around user considerations. For example, I let the user have more control over their account customization as they are allowed to change their profile picture as they please.

Another consideration I did was give the user more control on the statistics page and also match to the real world at the same time. When on the statistics page, I matched to the real world by making selected buttons green and non-selected buttons gray. The color green in the real world is typically used in places like traffic lights that mean go. Gray on the other hand symbolises things that are dead. Gray is also the most common color out of this website, that is used in many other real world applications to show that something is de-selected.

Functionality

Functionality is by far the most important implication. Functionality means that the website is to function as intended with no errors. To make sure to keep to this relevant implication, I will implement various techniques with programming and visual design. In short, the website has to deliver its promise to the user.

When signing in or creating an account, I will make sure to handle duplicates of accounts names. Since accounts should be unique, when a user first creates an account, I will give an error message if the account is already taken. When signing in, I will tell the user when they get some credentials incorrect, with a message saying that their account username or password doesn't match.

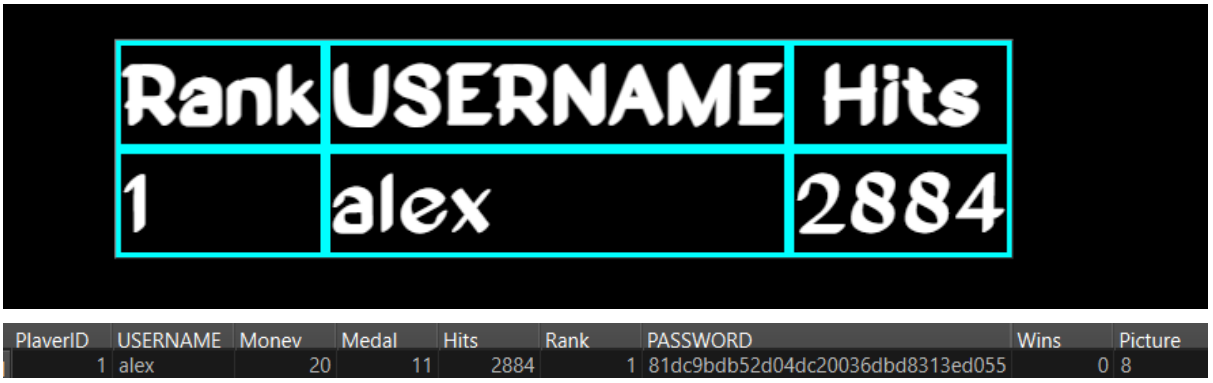
The game keeps track of user inputs and feeds it back into the database ready to be seen on the world's statistics page. The world statistics page or the personal statistics under the account page can show the user this data in a simple format. The whole website game keeps track of data live as promised by a live service game.

Website Data Integrity & Testing

When developing the website, it is important to make sure that what the user sees is what is proper to what the user wants. The most important part of the website for integrity is the filtering statistics page where the user has a numerous amount of inputs that generates an sql query from their inputs.

Searching By Name

For the search box, instead of using the filtering buttons, it will search for a specific username and ignore any of the filter options.



PlaverID	USERNAME	Monev	Medal	Hits	Rank	PASSWORD	Wins	Picture
1	alex	20	11	2884	1	81dc9bdb52d04dc20036dbd8313ed055	0	8

When searching for the username ‘alex’, we can see that the account matches the search. The rank, amount of hits and name are all displayed properly.

Filtering

When using the advanced filter options, I had to incorporate more complex query generation where I included ‘many to many’ tables and ‘Join’ tables into the query.

Let's start with a simple filtering test that incorporates the many to many table and foreign key join table.

Here I select the filtering option to find any user that has the ‘Styrofoam’ medal that is stored in the Join table. The ‘Noob’ selected button uses a Many to Many table and finds any user that has been awarded the Noob achievement.

The resulting table that gets generated is:

Rank	USERNAME	Hits
2	INSERT	328
3	hi_there	152
4	22197	0
5	22343	0
6	22177	0
7	22146	0
8	Aaronatedih	0

The query that we generated is:

```
FINAL: Select UserData.rank, UserData.USERNAME, UserData.hits From UserData Where (UserData.Medal = 0) And  ( PlayerID In (Select PlayerID Fro
m Awarded Where AwardID = (Select ID From Achievements Where Achievement = "Noob")))ORDER BY UserData.rank ASC;
```

Now we can see that out of all 8 accounts that exist in the database, only 7 of them were displayed. When we check the database to see why this has happened, we can see that the account ‘alex’ indeed is the only account that doesn't have this rank.

Main Database Table:

	PlaverID	USERNAME	Monev	Medal	Hits	Rank	PASSWORD	Wins	Picture
1	1	alex	20	11	2884	1	81dc9bdb52d04dc20036dbd8313ed055	0	8
2	2	INSERT	20	0	328	2	e10adc3949ba59abbe56e057f20f883e	0	8
3	3	22197	20	0	0	4	cf9bb12aa47ab3e8ce353b001b8b79e4	0	0
4	4	22343	20	0	0	5	91f724a7a4de4b13f12e65ce67ed0e23	0	0
5	5	22177	20	0	0	6	e10adc3949ba59abbe56e057f20f883e	0	0
6	6	22146	20	0	0	7	358aaf4a91006722f704e432b82fa60e	0	0
7	7	hi_there	20	0	152	3	e10adc3949ba59abbe56e057f20f883e	0	4
8	8	Aaronatedih	20	0	0	8	cb9fbb82698d4d941edb1db8c6796fe7	0	1

The achievements database table:

	ID	Achievement	Info
1	0	Starter	Discover the game for the first time
2	1	Loser	Lose your first game
3	3	Goat	Rank first
4	2	Finisher	Finish the game for the first time
5	4	Noob	Create an account

The Many to Many database connection table:

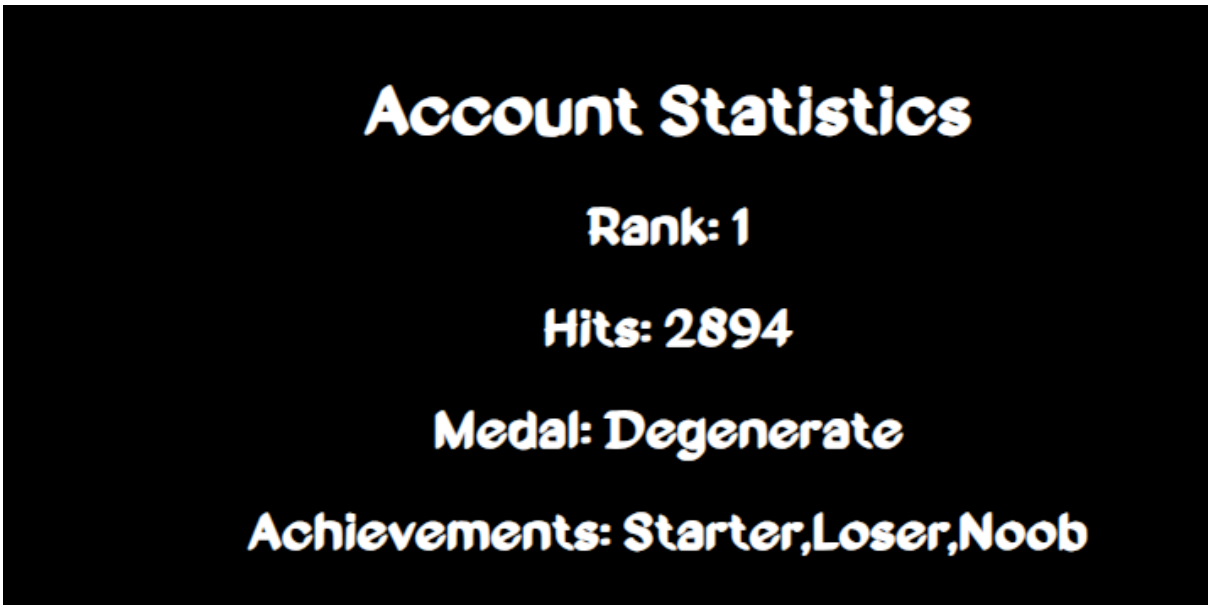
	AwardID	PlaverID
1	0	1
2	4	2
3	4	3
4	4	4
5	1	1
6	4	5
7	4	6
8	4	7
9	1	7
10	4	8

We know that the player id for “alex” is id 1 and that the id for the noob achievement is 4. We can see that yes, the table does not have any match for PlayerID 1 to any AwardID 4. Every other account (we can count 7 fours so seven accounts have the 4 awardID) has the achievement while the ‘alex account doesn't. So we can confirm that the data displayed is correct.

Personal Statistics

To test personal statistics, I am going to compare the account ‘alex’ with what we see on the front end.

Front End Statistics

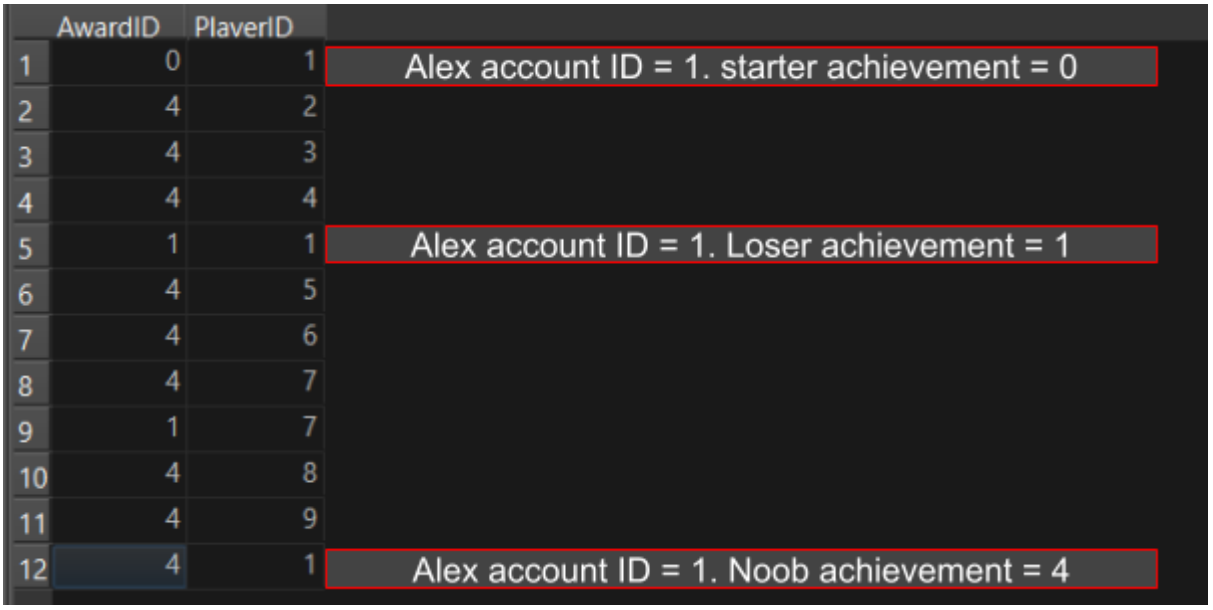


From what we see on this page, the database should have the degenerate medal underneath the account and the 3 achievements as well under the same account as well as the rank and hit amount.

Achievements Database Table

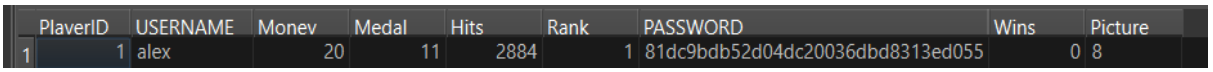
	ID	Achievement	Info
1	0	Starter	Discover the game for the first time
2	1	Loser	Lose your first game
3	3	Goat	Rank first
4	2	Finisher	Finish the game for the first time
5	4	Noob	Create an account

Awarded Database Table (The many to many table)



	AwardID	PlayerID	
1	0	1	Alex account ID = 1. starter achievement = 0
2	4	2	
3	4	3	
4	4	4	
5	1	1	Alex account ID = 1. Loser achievement = 1
6	4	5	
7	4	6	
8	4	7	
9	1	7	
10	4	8	
11	4	9	
12	4	1	Alex account ID = 1. Noob achievement = 4

UserData Main Database Table

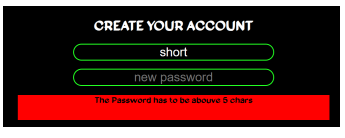


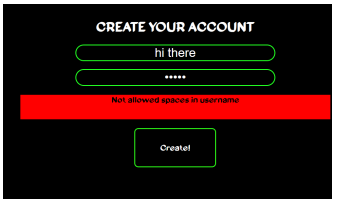
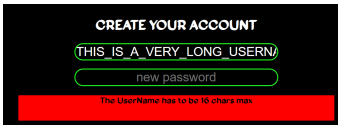
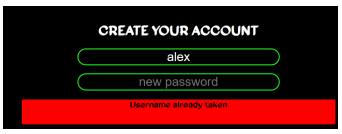
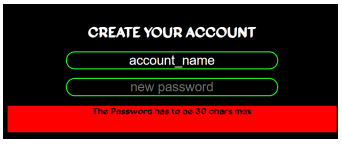
	PlayerID	USERNAME	Monev	Medal	Hits	Rank	PASSWORD	Wins	Picture
1	1	alex	20	11	2884	1	81dc9bdb52d04dc20036dbd8313ed055	0	8

We can see that everything matches up and that the data displayed on the front end matches what's stored in the backends database.

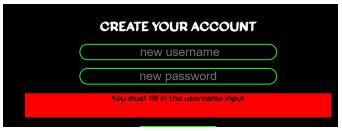
Other General Testing

Boundary Sign up page testing

Query / purpose	**Expected**	**Got**	**Problem / conclusion**
Username: short Password: 1234 Testing to see if the program catches the minimum password length	Password has to be above 5 characters		Passed

Username: hi there Password: 123456	To deny sign up because there is a detected space		Passed
Username: THIS_IS_A_VERY_LONG_USERNAME Password: 123456	Should deny account creation because the username Is over 16 characters		Passed
Username: alex Password: 123456	Testing for duplicates. Should deny duplicate password.		Passed
Username:account_name Password: 12345678901234567890123456789012345678 90	Testing password length		Passed

Non-Boundry Sign up page testing

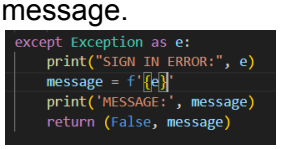
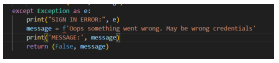
Querey	**Expected**	**Got**	**Problem / conclusion**
Username: Insert password : 123456	Successfully signed in without breaking or injecting	Signed in and created account: 	Passed
Username: Password:	Testing for a blank input		Passed

Boundary Sign in page testing

Querey	**Expected**	**Got**	**Problem / conclusion**

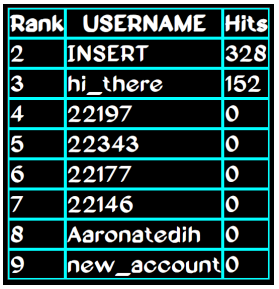
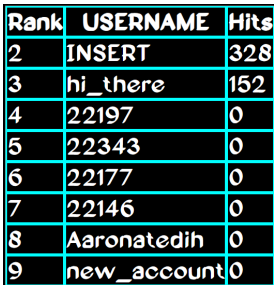
Username: INSERT password : 123456	To let you through and sign in	Signed in successfully	Passes
Username: INSERT Password: 12345	Fail, password is 123456 not 12345	Failed	Passes
UserName: notarealaccount Password: 123456	Fail, username doesn't exist	Failed	Passes

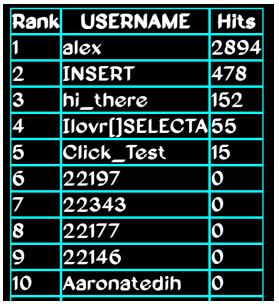
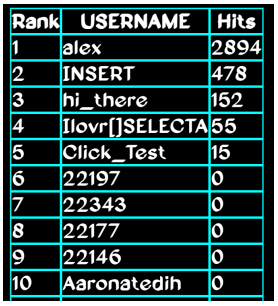
Non-Boundry Sign in page testing

Querey	**Expected**	**Got**	**Problem / conclusion**
Username: alex Password: 123456 The password does not match this username but is the password of the 'INSERT' account	Failure to sign in		Passes We were checking to see if the password system just checked for a valid password but ignored account matching.
Username: INSERT INTO password : 123456	Failed to sign in		Didn't sign in but had a bad error message. 
Username: INSERT INTO password : 123456	Failed to sign in		Passes New Code: 

Filtering Non-Boundry Testing

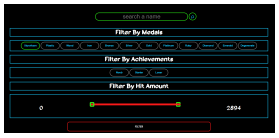
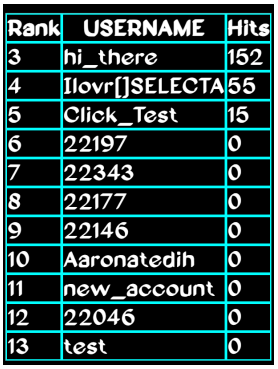
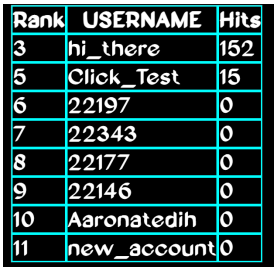
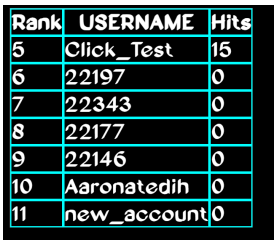
Change	Expected	Got	Problem / conclusion

Part:
 
Before:
`'medal'`
After:
`'meda3l'`
Changing html data on website on a filter button to not match intended purpose
	
	Rank USERNAME Hits
	2 INSERT 328
	3 hi_there 152
	4 22197 0
	5 22343 0
	6 22177 0
	7 22146 0
	8 Aaronatedih 0
	9 new_account 0
		
		Rank USERNAME Hits
		2 INSERT 328
		3 hi_there 152
		4 22197 0
		5 22343 0
		6 22177 0
		7 22146 0
		8 Aaronatedih 0
		9 new_account 0
			Passes
			
			We were checking if public site manipulation could affect results and it passed and didn't break the backend.

Empty filtering options  	 Rank USERNAME Hits 1 alex 2894 2 INSERT 478 3 hi_there 152 4 Ilovr[]SELECTA 55 5 Click_Test 15 6 22197 0 7 22343 0 8 22177 0 9 22146 0 10 Aaronatedih 0	 Rank USERNAME Hits 1 alex 2894 2 INSERT 478 3 hi_there 152 4 Ilovr[]SELECTA 55 5 Click_Test 15 6 22197 0 7 22343 0 8 22177 0 9 22146 0 10 Aaronatedih 0	Passed as i have a check to see if any filter option was selected. If not, will just get every user and their rank.
 	Nothing	 Rank USERNAME Hits	An attempt to inject by failed and there was no user in the database that went by than username

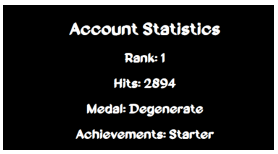
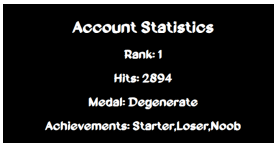
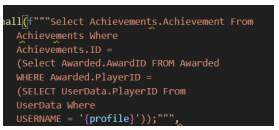
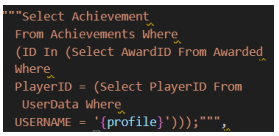
Filtering Boundary Testing

Change	Expected	Got	Problem / conclusion

	Everyone that has the styrofoam rank	 Rank USERNAME Hits 3 hi_there 152 4 Ilovr[]SELECTA 55 5 Click_Test 15 6 22197 0 7 22343 0 8 22177 0 9 22146 0 10 Aaronatedih 0 11 new_account 0 12 22046 0 13 test 0	Passed
	Everyone that has the noob achievement and styrofoam rank	 Rank USERNAME Hits 3 hi_there 152 5 Click_Test 15 6 22197 0 7 22343 0 8 22177 0 9 22146 0 10 Aaronatedih 0 11 new_account 0	Passed
	Everyone that has the styrofoam medal, noob achievement and a hit range of 0 to 143.	 Rank USERNAME Hits 5 Click_Test 15 6 22197 0 7 22343 0 8 22177 0 9 22146 0 10 Aaronatedih 0 11 new_account 0	Passed
	Everyone that has the Diamond Medal	 Rank USERNAME Hits	Passed No one has that medal

Account Statistics Display Tests

Test	**Expected**	**Got**	**Problem / conclusion**

Displaying personal statistics in	Get multiple achievements on account		The account has more than one achievement but isn't showing all achievements. Only one.
Displaying Personal statistics using a different query system	Get multiple achievements on account		Changed the query from this:  To this: 

404 Game Testing

Test	Passed / Failed	Notes
Rock Collisions	Passed Player received damage	We test to see if the player collides with the rock obstacles flying from the side of the screen
Mouse Missile Collision tests	Passed Player received damage	The missiles have to touch the mouse and the player then gets damaged.
Boss collision tests	Passed Player received damage	The bosses hitbox is smaller than it looks because I got feedback that the game was too hard so I nerfed the size of the hitbox.

Route testing

Route	Expected	Got	Problem /

			conclusion / notes
/	Home page	Home page	passed
/home	Home page	Home page	Passed
/hd8um923q1xdyhmn897	Game 404 page	Game 404 error page	The whole gimmick of the website is that it is a 404 error page boss battle so it made sense to make every 404 route detected by flask go to the game page.
/statistics	Statistics page	Statistics page	passed
/about	About page	About Page	passed
/create_account	Create account page	Create account page	passed
/sign_in_page	Sign in page	Sign in page	passed
/home#4	Home page	Home Page	Passed
/home/4	404 game page	404 Game page	Passed Since there is no page such as that route, we get rerouted to the 404 game page.